

new/usr/src/cmd/zoneadmd/zoneadmd.c

```
*****
60359 Wed Jan 30 11:17:45 2019
new/usr/src/cmd/zoneadmd/zoneadmd.c
10141 smatch fix for zoneadmd
*****  
_____ unchanged_portion_omitted _____
```

```
1827 int
1828 main(int argc, char *argv[])
1829 {
1830     int opt;
1831     zoneid_t zid;
1832     priv_set_t *privset;
1833     zone_state_t zstate;
1834     char parents_locale[MAXPATHLEN];
1835     brand_handle_t bh;
1836     int err;
1837
1838     pid_t pid;
1839     sigset_t blockset;
1840     sigset_t block_cld;
1841
1842     struct {
1843         sema_t sem;
1844         int status;
1845         zlog_t log;
1846     } *shstate;
1847     size_t shstatelen = getpagesize();
1848
1849     zlog_t errlog;
1850     zlog_t *zlogp;
1851
1852     int ctfd;
1853
1854     progrname = get_execbasename(argv[0]);
1855
1856     /*
1857      * Make sure stderr is unbuffered
1858      */
1859     (void) setbuffer(stderr, NULL, 0);
1860
1861     /*
1862      * Get out of the way of mounted filesystems, since we will daemonize
1863      * soon.
1864      */
1865     (void) chdir("/");
1866
1867     /*
1868      * Use the default system umask per PSARC 1998/110 rather than
1869      * anything that may have been set by the caller.
1870      */
1871     (void) umask(CMASK);
1872
1873     /*
1874      * Initially we want to use our parent's locale.
1875      */
1876     (void) setlocale(LC_ALL, "");
1877     (void) textdomain(TEXT_DOMAIN);
1878     (void) strlcpy(parents_locale, setlocale(LC_MESSAGES, NULL),
1879                   sizeof (parents_locale));
1880
1881     /*
1882      * This zlog_t is used for writing to stderr
1883      */
1884     errlog.logfile = stderr;
1885     errlog.buflen = errlog.loglen = 0;
```

1

new/usr/src/cmd/zoneadmd/zoneadmd.c

```
1886     errlog.buf = errlog.log = NULL;
1887     errlog.locale = parents_locale;
1888
1889     /*
1890      * We start off writing to stderr until we're ready to daemonize.
1891      */
1892     zlogp = &errlog;
1893
1894     /*
1895      * Process options.
1896      */
1897     while ((opt = getopt(argc, argv, "R:z:")) != EOF) {
1898         switch (opt) {
1899             case 'R':
1900                 zonecfg_set_root(optarg);
1901                 break;
1902             case 'z':
1903                 zone_name = optarg;
1904                 break;
1905             default:
1906                 usage();
1907         }
1908     }
1909
1910     if (zone_name == NULL)
1911         usage();
1912
1913     /*
1914      * Because usage() prints directly to stderr, it has gettext()
1915      * wrapping, which depends on the locale. But since zerror() calls
1916      * localize() which tweaks the locale, it is not safe to call zerror()
1917      * until after the last call to usage(). Fortunately, the last call
1918      * to usage() is just above and the first call to zerror() is just
1919      * below. Don't mess this up.
1920      */
1921     if (strcmp(zone_name, GLOBAL_ZONENAME) == 0) {
1922         zerror(zlogp, B_FALSE, "cannot manage the %s zone",
1923                GLOBAL_ZONENAME);
1924         return (1);
1925     }
1926
1927     if (zone_get_id(zone_name, &zid) != 0) {
1928         zerror(zlogp, B_FALSE, "could not manage %s: %s", zone_name,
1929                zonecfg_strerror(Z_NO_ZONE));
1930         return (1);
1931     }
1932
1933     if ((err = zone_get_state(zone_name, &zstate)) != Z_OK) {
1934         zerror(zlogp, B_FALSE, "failed to get zone state: %s",
1935                zonecfg_strerror(err));
1936         return (1);
1937     }
1938     if (zstate < ZONE_STATE_INCOMPLETE) {
1939         zerror(zlogp, B_FALSE,
1940               "cannot manage a zone which is in state '%s'",
1941               zone_state_str(zstate));
1942         return (1);
1943     }
1944
1945     if (zonecfg_default_brand(default_brand,
1946                               sizeof (default_brand)) != Z_OK) {
1947         zerror(zlogp, B_FALSE, "unable to determine default brand");
1948         return (1);
1949     }
1950
1951     /* Get a handle to the brand info for this zone */
```

2

```

1952     if (zone_get_brand(zone_name, brand_name, sizeof (brand_name))
1953         != Z_OK) {
1954         zerror(zlogp, B_FALSE, "unable to determine zone brand");
1955         return (1);
1956     }
1957     zone_isnative = (strcmp(brand_name, NATIVE_BRAND_NAME) == 0);
1958     zone_islabeled = (strcmp(brand_name, LABELED_BRAND_NAME) == 0);

1959     /*
1960      * In the alternate root environment, the only supported
1961      * operations are mount and umount. In this case, just treat
1962      * the zone as native if it is cluster. Cluster zones can be
1963      * native for the purpose of LU or upgrade, and the cluster
1964      * brand may not exist in the miniroot (such as in net install
1965      * upgrade).
1966      */
1967     if (strcmp(brand_name, CLUSTER_BRAND_NAME) == 0) {
1968         zone_iscluster = B_TRUE;
1969         if (zonetcfg_in_alt_root()) {
1970             (void) strlcpy(brand_name, default_brand,
1971                           sizeof (brand_name));
1972         }
1973     } else {
1974         zone_iscluster = B_FALSE;
1975     }

1976     if ((bh = brand_open(brand_name)) == NULL) {
1977         zerror(zlogp, B_FALSE, "unable to open zone brand");
1978         return (1);
1979     }

1980     /* Get state change brand hooks. */
1981     if (brand_callback_init(bh, zone_name) == -1) {
1982         zerror(zlogp, B_TRUE,
1983               "failed to initialize brand state change hooks");
1984         brand_close(bh);
1985         return (1);
1986     }

1987     brand_close(bh);

1988     /*
1989      * Check that we have all privileges. It would be nice to pare
1990      * this down, but this is at least a first cut.
1991      */
1992     if ((privset = priv_allocset()) == NULL) {
1993         zerror(zlogp, B_TRUE, "%s failed", "priv_allocset");
1994         return (1);
1995     }

1996     if (getppriv(PRIV_EFFECTIVE, privset) != 0) {
1997         zerror(zlogp, B_TRUE, "%s failed", "getppriv");
1998         priv_freeset(privset);
1999         return (1);
2000     }

2001     if (priv_isfullset(privset) == B_FALSE) {
2002         zerror(zlogp, B_FALSE, "You lack sufficient privilege to "
2003               "run this command (all privs required)");
2004         priv_freeset(privset);
2005         return (1);
2006     }

2007     priv_freeset(privset);

2008     if (mkzonedir(zlogp) != 0)
2009         return (1);

```

```

4

2019     /*
2020      * Pre-fork: setup shared state
2021      */
2022     if ((shstate = (void *)mmap(NULL, shstatelen,
2023                               PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, -1, (off_t)0)) ==
2024         MAP_FAILED) {
2025         zerror(zlogp, B_TRUE, "%s failed", "mmap");
2026         return (1);
2027     }
2028     if (sema_init(&shstate->sem, 0, USYNC_PROCESS, NULL) != 0) {
2029         zerror(zlogp, B_TRUE, "%s failed", "sema_init()");
2030         (void) munmap((char *)shstate, shstatelen);
2031         return (1);
2032     }
2033     shstate->log.logfile = NULL;
2034     shstate->log.buflen = shstatelen - sizeof (*shstate);
2035     shstate->log.loglen = shstate->log.buflen;
2036     shstate->log.buf = (char *)shstate + sizeof (*shstate);
2037     shstate->log.log = shstate->log.buf;
2038     shstate->log.locale = parents_locale;
2039     shstate->status = -1;

2040     /*
2041      * We need a SIGCHLD handler so the sema_wait() below will wake
2042      * up if the child dies without doing a sema_post().
2043      */
2044     (void) sigset(SIGCHLD, sigchld);
2045     /*
2046      * We must mask SIGCHLD until after we've coped with the fork
2047      * sufficiently to deal with it; otherwise we can race and
2048      * receive the signal before pid has been initialized
2049      * (yes, this really happens).
2050      */
2051     (void) sigemptyset(&block_cld);
2052     (void) sigaddset(&block_cld, SIGCHLD);
2053     (void) sigprocmask(SIG_BLOCK, &block_cld, NULL);

2054     /*
2055      * The parent only needs stderr after the fork, so close other fd's
2056      * that we inherited from zoneadm so that the parent doesn't have those
2057      * open while waiting. The child will close the rest after the fork.
2058      */
2059     closefrom(3);

2060     if ((ctfd = init_template()) == -1) {
2061         zerror(zlogp, B_TRUE, "failed to create contract");
2062         return (1);
2063     }

2064     /*
2065      * Do not let another thread localize a message while we are forking.
2066      */
2067     (void) mutex_lock(&msglock);
2068     pid = fork();
2069     (void) mutex_unlock(&msglock);

2070     /*
2071      * In all cases (parent, child, and in the event of an error) we
2072      * don't want to cause creation of contracts on subsequent fork()'s.
2073      */
2074     (void) ct_tmpl_clear(ctfd);
2075     (void) close(ctfd);

2076     if (pid == -1) {
2077         zerror(zlogp, B_TRUE, "could not fork");
2078     }

```

```

2084         return (1);
2085
2086     } else if (pid > 0) { /* parent */
2087         /* This marks a window of vulnerability in which we receive
2088         * the SIGCLD before falling into sema_wait (normally we would
2089         * get woken up from sema_wait with EINTR upon receipt of
2090         * SIGCLD). So we may need to use some other scheme like
2091         * sema_posting in the sigcld handler.
2092         * blech
2093         */
2094         (void) sema_wait(&shstate->sem);
2095         (void) sema_destroy(&shstate->sem);
2096         if (shstate->status != 0)
2097             (void) waitpid(pid, NULL, WNOHANG);
2098
2099         /* It's ok if we die with SIGPIPE. It's not like we could have
2100         * done anything about it.
2101         */
2102         (void) fprintf(stderr, "%s", shstate->log.buf);
2103         _exit(shstate->status == 0 ? 0 : 1);
2104     }
2105
2106     /*
2107      * The child charges on.
2108      */
2109     (void) sigset(SIGCHLD, SIG_DFL);
2110     (void) sigprocmask(SIG_UNBLOCK, &block_cld, NULL);
2111
2112     /*
2113      * SIGPIPE can be delivered if we write to a socket for which the
2114      * peer endpoint is gone. That can lead to too-early termination
2115      * of zoneadmd, and that's not good eats.
2116      */
2117     (void) sigset(SIGPIPE, SIG_IGN);
2118
2119     /*
2120      * Stop using stderr
2121      */
2122     zlogp = &shstate->log;
2123
2124     /*
2125      * We don't need stdout/stderr from now on.
2126      */
2127     closefrom(0);
2128
2129     /*
2130      * Initialize the syslog zlog_t. This needs to be done after
2131      * the call to closefrom().
2132      */
2133     logsys.buf = logsys.log = NULL;
2134     logsys.buflen = logsys.loglen = 0;
2135     logsys.logfile = NULL;
2136     logsys.locale = DEFAULT_LOCALE;
2137
2138     openlog("zoneadmd", LOG_PID, LOG_DAEMON);
2139
2140     /*
2141      * The eventstream is used to publish state changes in the zone
2142      * from the door threads to the console I/O poller.
2143      */
2144     if (eventstream_init() == -1) {
2145         zerror(zlogp, B_TRUE, "unable to create eventstream");
2146         goto child_out;
2147     }
2148

```

```

2150     (void) sprintf(zone_door_path, sizeof(zone_door_path),
2151                     "%s" ZONE_DOOR_PATH, zonecfg_get_root(), zone_name);
2152
2153     /*
2154      * See if another zoneadmd is running for this zone. If not, then we
2155      * can now modify system state.
2156      */
2157     if (make_daemon_exclusive(zlogp) == -1)
2158         goto child_out;
2159
2160     /*
2161      * Create/join a new session; we need to be careful of what we do with
2162      * the console from now on so we don't end up being the session leader
2163      * for the terminal we're going to be handing out.
2164      */
2165     (void) setsid();
2166
2167     /*
2168      * This thread shouldn't be receiving any signals; in particular,
2169      * SIGCHLD should be received by the thread doing the fork().
2170      */
2171     (void) sigfillset(&blockset);
2172     (void) thr_sigsetmask(SIG_BLOCK, &blockset, NULL);
2173
2174     /*
2175      * Setup the console device and get ready to serve the console;
2176      * once this has completed, we're ready to let console clients
2177      * make an attempt to connect (they will block until
2178      * serve_console_sock() below gets called, and any pending
2179      * connection is accept()ed).
2180      */
2181     if (!zonecfg_in_alt_root() && init_console(zlogp) < 0)
2182         goto child_out;
2183
2184     /*
2185      * Take the lock now, so that when the door server gets going, we
2186      * are guaranteed that it won't take a request until we are sure
2187      * that everything is completely set up. See the child_out: label
2188      * below to see why this matters.
2189      */
2190     (void) mutex_lock(&lock);
2191
2192     /*
2193      * Init semaphore for scratch zones. */
2194     if (sema_init(&scratch_sem, 0, USYNC_THREAD, NULL) == -1) {
2195         zerror(zlogp, B_TRUE,
2196                 "failed to initialize semaphore for scratch zone");
2197         goto child_out;
2198     }
2199
2200     /*
2201      * open the dladm handle */
2202     if (dladm_open(&dld_handle) != DLADM_STATUS_OK) {
2203         zerror(zlogp, B_FALSE, "failed to open dladm handle");
2204         goto child_out;
2205     }
2206
2207     /*
2208      * Note: door setup must occur *after* the console is setup.
2209      * This is so that as zlogin tests the door to see if zoneadmd
2210      * is ready yet, we know that the console will get serviced
2211      * once door_info() indicates that the door is "up".
2212      */
2213     if (setup_door(zlogp) == -1)
2214         goto child_out;
2215
2216     /*
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
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
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
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
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
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
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
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
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
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
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
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
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
3144
3145
3146
3147
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
3188
3188
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
3229
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3239
3240
3241
3242
3243
3244
3245
3246
3247
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
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
3329
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3339
3340
3341
3342
3343
3344
3345
3346
3347
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
3388
3388
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3398
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3489
3490
3491
3492
3493
3494
3495
3496
3497
3497
3498
3499
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
3549
3549
3550
3551
3552
3553
3554
3555
3556
3557
3558
3559
3559
3560
3561
3562
3563
3564
3565
3566
3567
3568
3569
3569
3570
3571
3572
3573
3574
3575
3576
3577
3578
3579
3579
3580
3581
3582
3583
3584
3585
3586
3587
3588
3589
3589
3590
3591
3592
3593
3594
3595
3596
3597
3597
3598
3599
3599
3600
3601
3602
3603
3604
3605
3606
3607
3608
3609
3609
3610
3611
3612
3613
3614
3615
3616
3617
3618
3619
3619
3620
3621
3622
3623
3624
3625
3626
3627
3628
3629
3629
3630
3631
3632
3633
3634
3635
3636
3637
3638
3639
3639
3640
3641
3642
3643
3644
3645
3646
3647
3648
3649
3649
3650
3651
3652
3653
3654
3655
3656
3657
3658
3659
3659
3660
3661
3662
3663
3664
3665
3666
3667
3668
3669
3669
3670
3671
3672
3673
3674
3675
3676
3677
3678
3679
3679
3680
3681
3682
3683
3684
3685
3686
3687
3688
3689
3689
3690
3691
3692
3693
3694
3695
3696
3697
3698
3698
3699
3700
3701
3702
3703
3704
3705
3706
3707
3708
3709
3709
3710
3711
3712
3713
3714
3715
3716
3717
3718
3719
3719
3720
3721
3722
3723
3724
3725
3726
3727
3728
3729
3729
3730
3731
3732
3733
3734
3735
3736
3737
3738
3739
3739
3740
3741
3742
3743
3744
3745
3746
3747
3748
3749
3749
3750
3751
3752
3753
3754
3755
3756
3757
3758
3759
3759
3760
3761
3762
3763
3764
3765
3766
3767
3768
3769
3769
3770
3771
3772
3773
3774
3775
3776
3777
3778
3779
3779
3780
3781
3782
3783
3784
3785
3786
3787
3788
3789
3789
3790
3791
3792
3793
3794
3795
3796
3797
3798
3798
3799
3800
3801
3802
3803
3804
3805
3806
3807
3808
3809
3809
3810
3811
3812
3813
3814
3815
3816
3817
3818
3819
3819
3820
3821
3822
3823
3824
3825
3826
3827
3828
3829
3829
3830
3831
3832
3833
3834
3835
3836
3837
3838
3839
3839
3840
3841
3842
3843
3844
3845
3846
3847
3848
3849
3849
3850
3851
3852
3853
3854
3855
3856
3857
3858
3859
3859
3860
3861
3862
3863
3864
3865
3866
3867
3868
3869
3869
3870
3871
3872
3873
3874
3875
3876
3877
3878
3879
3879
3880
3881
3882
3883
3884
3885
3886
3887
3888
3889
3889
3890
3891
3892
3893
3894
3895
3896
3897
3898
3898
3899
3900
3901
3902
3903
3904
3905
3906
3907
3908
3909
3909
3910
3911
3912
3913
3914
3915
3916
3917
3918
3919
3919
3920
3921
3922
3923
3924
3925
3926
3927
3928
3929
3929
3930
3931
3932
3933
3934
3935
3936
3937
3938
3939
3939
3940
3941
3942
3943
3944
3945
3946
3947
3948
3949
3949
3950
3951
3952
3953
3954
3955
3956
3957
3958
395
```

```

2216     * Things seem OK so far; tell the parent process that we're done
2217     * with setup tasks. This will cause the parent to exit, signalling
2218     * to zoneadm, zlogin, or whatever forked it that we are ready to
2219     * service requests.
2220     */
2221     shstate->status = 0;
2222     (void) sema_post(&shstate->sem);
2223     (void) munmap((char *)shstate, shstatelen);
2224     shstate = NULL;
2225
2226     (void) mutex_unlock(&lock);
2227
2228     /*
2229     * zlogp is now invalid, so reset it to the syslog logger.
2230     */
2231     zlogp = &logsys;
2232
2233     /*
2234     * Now that we are free of any parents, switch to the default locale.
2235     */
2236     (void) setlocale(LC_ALL, DEFAULT_LOCALE);
2237
2238     /*
2239     * At this point the setup portion of main() is basically done, so
2240     * we reuse this thread to manage the zone console. When
2241     * serve_console() has returned, we are past the point of no return
2242     * in the life of this zoneadmd.
2243     */
2244     if (zonecfg_in_alt_root()) {
2245         /*
2246         * This is just awful, but mounted scratch zones don't (and
2247         * can't) have consoles. We just wait for unmount instead.
2248         */
2249         while (sema_wait(&scratch_sem) == EINTR)
2250             ;
2251     } else {
2252         serve_console(zlogp);
2253         assert(in_death_throes);
2254     }
2255
2256     /*
2257     * This is the next-to-last part of the exit interlock. Upon calling
2258     * fdetach(), the door will go unreferenced; once any
2259     * outstanding requests (like the door thread doing Z_HALT) are
2260     * done, the door will get an UNREF notification; when it handles
2261     * the UNREF, the door server will cause the exit. It's possible
2262     * that fdetach() can fail because the file is in use, in which
2263     * case we'll retry the operation.
2264     */
2265     assert(!MUTEX_HELD(&lock));
2266     for (;;) {
2267         if ((fdetach(zone_door_path) == 0) || (errno != EBUSY))
2268             break;
2269         yield();
2270     }
2271     for (;;) {
2272         (void) pause();
2273     }
2274
2275 child_out:
2276     assert(pid == 0);
2277
2278     if (shstate != NULL) {
2279         shstate->status = -1;
2280         (void) sema_post(&shstate->sem);
2281         (void) munmap((char *)shstate, shstatelen);

```

```

2281     }
2282     /*
2283     * This might trigger an unref notification, but if so,
2284     * we are still holding the lock, so our call to exit will
2285     * ultimately win the race and will publish the right exit
2286     * code.
2287     */
2288     if (zone_door != -1) {
2289         assert(MUTEX_HELD(&lock));
2290         (void) door_revoke(zone_door);
2291         (void) fdetach(zone_door_path);
2292     }
2293     if (dld_handle != NULL)
2294         dladm_close(dld_handle);
2295
2296     return (1); /* return from main() forcibly exits an MT process */
2297 }
2298
2299 unchanged_portion_omitted

```