```
**********************************************************
   12676 Thu Jun 30 21:58:38 2016
new/usr/src/cmd/sgs/elfdump/common/struct_layout_sparc.c
Code review comments from pmooney (sundry), and igork (screwups in zonecfg refac
**********************************************************
______unchanged_portion_omitted_


379 static const sl_prsecflags_layout_t prsecflags_layout = {
380         { 0,    20,     0,      0 },            /* sizeof (prsecflags_t) */
381         { 0,    4,      0,      0 },            /* pr_version */
382         { 4,    4,      0,      0 },            /* pr_effective */
383         { 8,    4,      0,      0 },            /* pr_inherit */
384         { 12,   4,      0,      0 },            /* pr_lower */
385         { 16,   4,      0,      0 },            /* pr_upper */
386 };


389 #endif /* ! codereview */


392 static const sl_arch_layout_t layout_sparc = {
393         &auxv_layout,
394         &fltset_layout,
395         &lwpsinfo_layout,
396         &lwpstatus_layout,
397         &prcred_layout,
398         &priv_impl_info_layout,
399         &prpriv_layout,
400         &psinfo_layout,
401         &pstatus_layout,
402         &prgregset_layout,
403         &prpsinfo_layout,
404         &prstatus_layout,
405         &sigaction_layout,
406         &siginfo_layout,
407         &sigset_layout,
408         &stack_layout,
409         &sysset_layout,
410         &timestruc_layout,
411         &utsname_layout,
412         &prfdinfo_layout,
413         &prsecflags_layout,
414 #endif /* ! codereview */
415 };


418 const sl_arch_layout_t *
419 struct_layout_sparc(void)
420 {
421         return (&layout_sparc);
422 }
```

****
12722 Thu Jun 30 21:58:39 2016
new/usr/src/cmd/sgs/elfdump/common/struct_layout_sparcv9.c
Code review comments from pmooney (sundry), and igork (screwups in zonecfg refac
****
_____***unchanged_portion_omitted_***

```
380 static const sl_prsecflags_layout_t prsecflags_layout = {
381         { 0,    20,     0,      0 },            /* sizeof (prsecflags_t) */
382         { 0,    4,      0,      0 },            /* pr_version */
383         { 4,    4,      0,      0 },            /* pr_effective */
384         { 8,    4,      0,      0 },            /* pr_inherit */
385         { 12,   4,      0,      0 },            /* pr_lower */
386         { 16,   4,      0,      0 },            /* pr_upper */
387 };


390 #endif /* ! codereview */


393 static const sl_arch_layout_t layout_sparcv9 = {
394         &auxv_layout,
395         &fltset_layout,
396         &lwpsinfo_layout,
397         &lwpstatus_layout,
398         &prcred_layout,
399         &priv_impl_info_layout,
400         &prpriv_layout,
401         &psinfo_layout,
402         &pstatus_layout,
403         &prgregset_layout,
404         &prpsinfo_layout,
405         &prstatus_layout,
406         &sigaction_layout,
407         &siginfo_layout,
408         &sigset_layout,
409         &stack_layout,
410         &psset_layout,
411         &timestruc_layout,
412         &utsname_layout,
413         &prfdinfo_layout,
414         &prsecflags_layout,
415 #endif /* ! codereview */
416 };
417  const sl_arch_layout_t *


382 const sl_arch_layout_t *
418 struct_layout_sparcv9(void)
419 {
420         return (&layout_sparcv9);
421 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  201398 Thu Jun 30 21:58:40 2016
new/usr/src/cmd/zonecfg/zonecfg.c
Code review comments from pmooney (sundry), and igork (screwups in zonecfg refac
**********************************************************
______unchanged_portion_omitted_

1834 void
1835 export_func(cmd_t *cmd)
1836 {
1837         struct zone_nwiftab nwiftab;
1838         struct zone_fstab fstab;
1839         struct zone_devtab devtab;
1840         struct zone_attrtab attrtab;
1841         struct zone_rctltab rctltab;
1842         struct zone_dstab dstab;
1843         struct zone_psettab psettab;
1844         struct zone_mcaptab mcaptab;
1845         struct zone_rctlvaltab *valptr;
1846         struct zone_admintab admintab;
1847         struct zone_secflagstab secflagstab;
1848         int err, arg;
1849         char zonepath[MAXPATHLEN], outfile[MAXPATHLEN], pool[MAXNAMELEN];
1850         char bootargs[BOOTARGS_MAX];
1851         char sched[MAXNAMELEN];
1852         char brand[MAXNAMELEN];
1853         char hostidp[HW_HOSTID_LEN];
1854         char fsallowedp[ZONE_FS_ALLOWED_MAX];
1855         char *limitpriv;
1856         FILE *of;
1857         boolean_t autoboot;
1858         zone_iptype_t iptype;
1859         boolean_t need_to_close = B_FALSE;
1860         boolean_t arg_err = B_FALSE;

1862         assert(cmd != NULL);

1864         outfile[0] = '\0';
1865         optind = 0;
1866         while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "?f:")) != EOF) {
1867                 switch (arg) {
1868                 case '?':
1869                         if (optopt == '?')
1870                                 longer_usage(CMD_EXPORT);
1871                         else
1872                                 short_usage(CMD_EXPORT);
1873                         arg_err = B_TRUE;
1874                         break;
1875                 case 'f':
1876                         (void) strlcpy(outfile, optarg, sizeof (outfile));
1877                         break;
1878                 default:
1879                         short_usage(CMD_EXPORT);
1880                         arg_err = B_TRUE;
1881                         break;
1882                 }
1883         }
1884         if (arg_err)
1885                 return;

1887         if (optind != cmd->cmd_argc) {
1888                 short_usage(CMD_EXPORT);
1889                 return;
1890         }
1891         if (strlen(outfile) == 0) {
1892                 of = stdout;
```



```
1893         } else {
1894                 if ((of = fopen(outfile, "w")) == NULL) {
1895                         zerr(gettext("opening file %s: %s"),
1896                             outfile, strerror(errno));
1897                         goto done;
1898                 }
1899                 setbuf(of, NULL);
1900                 need_to_close = B_TRUE;
1901         }

1903         if ((err = initialize(B_TRUE)) != Z_OK)
1904                 goto done;

1906         (void) fprintf(of, "%s -b\n", cmd_to_str(CMD_CREATE));

1908         if (zonecfg_get_zonepath(handle, zonepath, sizeof (zonepath)) == Z_OK &&
1909             strlen(zonepath) > 0)
1910                 (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
1911                     pt_to_str(PT_ZONEPATH), zonepath);

1913         if ((zone_get_brand(zone, brand, sizeof (brand)) == Z_OK) &&
1914             (strcmp(brand, NATIVE_BRAND_NAME) != 0))
1915                 (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
1916                     pt_to_str(PT_BRAND), brand);

1918         if (zonecfg_get_autoboot(handle, &autoboot) == Z_OK)
1919                 (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
1920                     pt_to_str(PT_AUTOBOOT), autoboot ? "true" : "false");

1922         if (zonecfg_get_bootargs(handle, bootargs, sizeof (bootargs)) == Z_OK &&
1923             strlen(bootargs) > 0) {
1924                 (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
1925                     pt_to_str(PT_BOOTARGS), bootargs);
1926         }

1928         if (zonecfg_get_pool(handle, pool, sizeof (pool)) == Z_OK &&
1929             strlen(pool) > 0)
1930                 (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
1931                     pt_to_str(PT_POOL), pool);

1933         if (zonecfg_get_limitpriv(handle, &limitpriv) == Z_OK &&
1934             strlen(limitpriv) > 0) {
1935                 (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
1936                     pt_to_str(PT_LIMITPRIV), limitpriv);
1937                 free(limitpriv);
1938         }

1940         if (zonecfg_get_sched_class(handle, sched, sizeof (sched)) == Z_OK &&
1941             strlen(sched) > 0)
1942                 (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
1943                     pt_to_str(PT_SCHED), sched);

1945         if (zonecfg_get_iptype(handle, &iptype) == Z_OK) {
1946                 switch (iptype) {
1947                 case ZS_SHARED:
1948                         (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
1949                             pt_to_str(PT_IPTYPE), "shared");
1950                         break;
1951                 case ZS_EXCLUSIVE:
1952                         (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
1953                             pt_to_str(PT_IPTYPE), "exclusive");
1954                         break;
1955                 }
1956         }

1958         if (zonecfg_get_hostid(handle, hostidp, sizeof (hostidp)) == Z_OK) {
```

```
1959                (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
1960                    pt_to_str(PT_HOSTID), hostidp);
1961        }

1963        if (zonecfg_get_fs_allowed(handle, fsallowedp,
1964            sizeof (fsallowedp)) == Z_OK) {
1965                (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
1966                    pt_to_str(PT_FS_ALLOWED), fsallowedp);
1967        }

1969        if ((err = zonecfg_setfsent(handle)) != Z_OK) {
1970                zone_perror(zone, err, B_FALSE);
1971                goto done;
1972        }
1973        while (zonecfg_getfsent(handle, &fstab) == Z_OK) {
1974                zone_fsopt_t *optptr;

1976                (void) fprintf(of, "%s %s\n", cmd_to_str(CMD_ADD),
1977                    rt_to_str(RT_FS));
1978                export_prop(of, PT_DIR, fstab.zone_fs_dir);
1979                export_prop(of, PT_SPECIAL, fstab.zone_fs_special);
1980                export_prop(of, PT_RAW, fstab.zone_fs_raw);
1981                export_prop(of, PT_TYPE, fstab.zone_fs_type);
1982                for (optptr = fstab.zone_fs_options; optptr != NULL;
1983                    optptr = optptr->zone_fsopt_next) {
1984                        /*
1985                         * Simple property values with embedded equal signs
1986                         * need to be quoted to prevent the lexer from
1987                         * mis-parsing them as complex name=value pairs.
1988                         */
1989                        if (strchr(optptr->zone_fsopt_opt, '='))
1990                                (void) fprintf(of, "%s %s \"%s\"\n",
1991                                    cmd_to_str(CMD_ADD),
1992                                    pt_to_str(PT_OPTIONS),
1993                                    optptr->zone_fsopt_opt);
1994                        else
1995                                (void) fprintf(of, "%s %s %s\n",
1996                                    cmd_to_str(CMD_ADD),
1997                                    pt_to_str(PT_OPTIONS),
1998                                    optptr->zone_fsopt_opt);
1999                }
2000                (void) fprintf(of, "%s\n", cmd_to_str(CMD_END));
2001                zonecfg_free_fs_option_list(fstab.zone_fs_options);
2002        }
2003        (void) zonecfg_endfsent(handle);

2005        if ((err = zonecfg_setnwifent(handle)) != Z_OK) {
2006                zone_perror(zone, err, B_FALSE);
2007                goto done;
2008        }
2009        while (zonecfg_getnwifent(handle, &nwiftab) == Z_OK) {
2010                (void) fprintf(of, "%s %s\n", cmd_to_str(CMD_ADD),
2011                    rt_to_str(RT_NET));
2012                export_prop(of, PT_ADDRESS, nwiftab.zone_nwif_address);
2013                export_prop(of, PT_ALLOWED_ADDRESS,
2014                    nwiftab.zone_nwif_allowed_address);
2015                export_prop(of, PT_PHYSICAL, nwiftab.zone_nwif_physical);
2016                export_prop(of, PT_DEFROUTER, nwiftab.zone_nwif_defrouter);
2017                (void) fprintf(of, "%s\n", cmd_to_str(CMD_END));
2018        }
2019        (void) zonecfg_endnwifent(handle);

2021        if ((err = zonecfg_setdevent(handle)) != Z_OK) {
2022                zone_perror(zone, err, B_FALSE);
2023                goto done;
2024        }
```



```
2025        while (zonecfg_getdevent(handle, &devtab) == Z_OK) {
2026                (void) fprintf(of, "%s %s\n", cmd_to_str(CMD_ADD),
2027                    rt_to_str(RT_DEVICE));
2028                export_prop(of, PT_MATCH, devtab.zone_dev_match);
2029                (void) fprintf(of, "%s\n", cmd_to_str(CMD_END));
2030        }
2031        (void) zonecfg_enddevent(handle);

2033        if (zonecfg_getmcapent(handle, &mcaptab) == Z_OK) {
2034                char buf[128];

2036                (void) fprintf(of, "%s %s\n", cmd_to_str(CMD_ADD),
2037                    rt_to_str(RT_MCAP));
2038                bytes_to_units(mcaptab.zone_physmem_cap, buf, sizeof (buf));
2039                (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
2040                    pt_to_str(PT_PHYSICAL), buf);
2041                (void) fprintf(of, "%s\n", cmd_to_str(CMD_END));
2042        }

2044        if ((err = zonecfg_setrctlent(handle)) != Z_OK) {
2045                zone_perror(zone, err, B_FALSE);
2046                goto done;
2047        }
2048        while (zonecfg_getrctlent(handle, &rctltab) == Z_OK) {
2049                (void) fprintf(of, "%s rctl\n", cmd_to_str(CMD_ADD));
2050                export_prop(of, PT_NAME, rctltab.zone_rctl_name);
2051                for (valptr = rctltab.zone_rctl_valptr; valptr != NULL;
2052                    valptr = valptr->zone_rctlval_next) {
2053                        fprintf(of, "%s %s (%s=%s,%s=%s,%s=%s)\n",
2054                            cmd_to_str(CMD_ADD), pt_to_str(PT_VALUE),
2055                            pt_to_str(PT_PRIV), valptr->zone_rctlval_priv,
2056                            pt_to_str(PT_LIMIT), valptr->zone_rctlval_limit,
2057                            pt_to_str(PT_ACTION), valptr->zone_rctlval_action);
2058                }
2059                (void) fprintf(of, "%s\n", cmd_to_str(CMD_END));
2060                zonecfg_free_rctl_value_list(rctltab.zone_rctl_valptr);
2061        }
2062        (void) zonecfg_endrctlent(handle);

2064        if ((err = zonecfg_setattrent(handle)) != Z_OK) {
2065                zone_perror(zone, err, B_FALSE);
2066                goto done;
2067        }
2068        while (zonecfg_getattrent(handle, &attrtab) == Z_OK) {
2069                (void) fprintf(of, "%s %s\n", cmd_to_str(CMD_ADD),
2070                    rt_to_str(RT_ATTR));
2071                export_prop(of, PT_NAME, attrtab.zone_attr_name);
2072                export_prop(of, PT_TYPE, attrtab.zone_attr_type);
2073                export_prop(of, PT_VALUE, attrtab.zone_attr_value);
2074                (void) fprintf(of, "%s\n", cmd_to_str(CMD_END));
2075        }
2076        (void) zonecfg_endattrent(handle);

2078        if ((err = zonecfg_setdsent(handle)) != Z_OK) {
2079                zone_perror(zone, err, B_FALSE);
2080                goto done;
2081        }
2082        while (zonecfg_getdsent(handle, &dstab) == Z_OK) {
2083                (void) fprintf(of, "%s %s\n", cmd_to_str(CMD_ADD),
2084                    rt_to_str(RT_DATASET));
2085                export_prop(of, PT_NAME, dstab.zone_dataset_name);
2086                (void) fprintf(of, "%s\n", cmd_to_str(CMD_END));
2087        }
2088        (void) zonecfg_enddsent(handle);

2090        if (zonecfg_getpsetent(handle, &psettab) == Z_OK) {
```

```
2091                 (void) fprintf(of, "%s %s\n", cmd_to_str(CMD_ADD),
2092                     rt_to_str(RT_DCPU));
2093                 if (strcmp(psettab.zone_ncpu_min, psettab.zone_ncpu_max) == 0)
2094                         (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
2095                             pt_to_str(PT_NCPUS), psettab.zone_ncpu_max);
2096                 else
2097                         (void) fprintf(of, "%s %s=%s-%s\n", cmd_to_str(CMD_SET),
2098                             pt_to_str(PT_NCPUS), psettab.zone_ncpu_min,
2099                             psettab.zone_ncpu_max);
2100                 if (psettab.zone_importance[0] != '\0')
2101                         (void) fprintf(of, "%s %s=%s\n", cmd_to_str(CMD_SET),
2102                             pt_to_str(PT_IMPORTANCE), psettab.zone_importance);
2103                 (void) fprintf(of, "%s\n", cmd_to_str(CMD_END));
2104         }

2106         if ((err = zonecfg_setadminent(handle)) != Z_OK) {
2107                 zone_perror(zone, err, B_FALSE);
2108                 goto done;
2109         }
2110         while (zonecfg_getadminent(handle, &admintab) == Z_OK) {
2111                 (void) fprintf(of, "%s %s\n", cmd_to_str(CMD_ADD),
2112                     rt_to_str(RT_ADMIN));
2113                 export_prop(of, PT_USER, admintab.zone_admin_user);
2114                 export_prop(of, PT_AUTHS, admintab.zone_admin_auths);
2115                 (void) fprintf(of, "%s\n", cmd_to_str(CMD_END));
2116         }

2118         (void) zonecfg_endadminent(handle);

2120         if (zonecfg_getsecflagsent(handle, &secflagstab) == Z_OK) {
2120         if ((err = zonecfg_getsecflagsent(handle, &secflagstab)) != Z_OK) {
2121                 zone_perror(zone, err, B_FALSE);
2122                 goto done;
2123         }

2121                 (void) fprintf(of, "%s %s\n", cmd_to_str(CMD_ADD),
2122                     rt_to_str(RT_SECFLAGS));
2123                 export_prop(of, PT_DEFAULT, secflagstab.zone_secflags_default);
2124                 export_prop(of, PT_LOWER, secflagstab.zone_secflags_lower);
2125                 export_prop(of, PT_UPPER, secflagstab.zone_secflags_upper);
2126                 (void) fprintf(of, "%s\n", cmd_to_str(CMD_END));
2127         }
2128 #endif /* ! codereview */

2130         /*
2131          * There is nothing to export for pcap since this resource is just
2132          * a container for an rctl alias.
2133          */

2135 done:
2136         if (need_to_close)
2137                 (void) fclose(of);
2138 }

2140 void
2141 exit_func(cmd_t *cmd)
2142 {
2143         int arg, answer;
2144         boolean_t arg_err = B_FALSE;

2146         optind = 0;
2147         while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "?F")) != EOF) {
2148                 switch (arg) {
2149                 case '?':
2150                         longer_usage(CMD_EXIT);
2151                         arg_err = B_TRUE;
```



```
2152                         break;
2153                 case 'F':
2154                         force_exit = B_TRUE;
2155                         break;
2156                 default:
2157                         short_usage(CMD_EXIT);
2158                         arg_err = B_TRUE;
2159                         break;
2160                 }
2161         }
2162         if (arg_err)
2163                 return;

2165         if (optind < cmd->cmd_argc) {
2166                 short_usage(CMD_EXIT);
2167                 return;
2168         }

2170         if (global_scope || force_exit) {
2171                 time_to_exit = B_TRUE;
2172                 return;
2173         }

2175         answer = ask_yesno(B_FALSE, "Resource incomplete; really quit");
2176         if (answer == -1) {
2177                 zerr(gettext("Resource incomplete, input "
2178                     "not from terminal and -F not specified:\n%s command "
2179                     "ignored, but exiting anyway."), cmd_to_str(CMD_EXIT));
2180                 exit(Z_ERR);
2181         } else if (answer == 1) {
2182                 time_to_exit = B_TRUE;
2183         }
2184         /* (answer == 0) => just return */
2185 }

2187 static int
2188 validate_zonepath_syntax(char *path)
2189 {
2190         if (path[0] != '/') {
2191                 zerr(gettext("%s is not an absolute path."), path);
2192                 return (Z_ERR);
2193         }
2194         /* If path is all slashes, then fail */
2195         if (strspn(path, "/") == strlen(path)) {
2196                 zerr(gettext("/ is not allowed as a %s."),
2197                     pt_to_str(PT_ZONEPATH));
2198                 return (Z_ERR);
2199         }
2200         return (Z_OK);
2201 }

2203 static void
2204 add_resource(cmd_t *cmd)
2205 {
2206         int type;
2207         struct zone_psettab tmp_psettab;
2208         struct zone_mcaptab tmp_mcaptab;
2209         struct zone_secflagstab tmp_secflagstab;
2210         uint64_t tmp;
2211         uint64_t tmp_mcap;
2212         char pool[MAXNAMELEN];

2214         if ((type = cmd->cmd_res_type) == RT_UNKNOWN) {
2215                 long_usage(CMD_ADD, B_TRUE);
2216                 goto bad;
2217         }
```

```
2219         switch (type) {
2220         case RT_FS:
2221                 bzero(&in_progress_fstab, sizeof (in_progress_fstab));
2222                 return;
2223         case RT_NET:
2224                 bzero(&in_progress_nwiftab, sizeof (in_progress_nwiftab));
2225                 return;
2226         case RT_DEVICE:
2227                 bzero(&in_progress_devtab, sizeof (in_progress_devtab));
2228                 return;
2229         case RT_RCTL:
2230                 if (global_zone)
2231                         zerr(gettext("WARNING: Setting a global zone resource "
2232                             "control too low could deny\nservice "
2233                             "to even the root user; "
2234                             "this could render the system impossible\n"
2235                             "to administer.  Please use caution."));
2236                 bzero(&in_progress_rctltab, sizeof (in_progress_rctltab));
2237                 return;
2238         case RT_ATTR:
2239                 bzero(&in_progress_attrtab, sizeof (in_progress_attrtab));
2240                 return;
2241         case RT_DATASET:
2242                 bzero(&in_progress_dstab, sizeof (in_progress_dstab));
2243                 return;
2244         case RT_DCPU:
2245                 /* Make sure there isn't already a cpu-set or cpu-cap entry. */
2246                 if (zonecfg_lookup_pset(handle, &tmp_psettab) == Z_OK) {
2247                         zerr(gettext("The %s resource already exists."),
2248                             rt_to_str(RT_DCPU));
2249                         goto bad;
2250                 }
2251                 if (zonecfg_get_aliased_rctl(handle, ALIAS_CPUCAP, &tmp) !=
2252                     Z_NO_ENTRY) {
2253                         zerr(gettext("The %s resource already exists."),
2254                             rt_to_str(RT_PCAP));
2255                         goto bad;
2256                 }

2258                 /* Make sure the pool property isn't set. */
2259                 if (zonecfg_get_pool(handle, pool, sizeof (pool)) == Z_OK &&
2260                     strlen(pool) > 0) {
2261                         zerr(gettext("The %s property is already set.  "
2262                             "A persistent pool is incompatible with\nthe %s "
2263                             "resource."),
2264                             pt_to_str(PT_POOL), rt_to_str(RT_DCPU));
2265                         goto bad;
2266                 }

2268                 bzero(&in_progress_psettab, sizeof (in_progress_psettab));
2269                 return;
2270         case RT_PCAP:
2271                 /*
2272                  * Make sure there isn't already a cpu-set or incompatible
2273                  * cpu-cap rctls.
2274                  */
2275                 if (zonecfg_lookup_pset(handle, &tmp_psettab) == Z_OK) {
2276                         zerr(gettext("The %s resource already exists."),
2277                             rt_to_str(RT_DCPU));
2278                         goto bad;
2279                 }

2281                 switch (zonecfg_get_aliased_rctl(handle, ALIAS_CPUCAP, &tmp)) {
2282                 case Z_ALIAS_DISALLOW:
2283                         zone_perror(rt_to_str(RT_PCAP), Z_ALIAS_DISALLOW,
```



```
2284                             B_FALSE);
2285                         goto bad;

2287                 case Z_OK:
2288                         zerr(gettext("The %s resource already exists."),
2289                             rt_to_str(RT_PCAP));
2290                         goto bad;

2292                 default:
2293                         break;
2294                 }
2295                 return;
2296         case RT_MCAP:
2297                 /*
2298                  * Make sure there isn't already a mem-cap entry or max-swap
2299                  * or max-locked rctl.
2300                  */
2301                 if (zonecfg_lookup_mcap(handle, &tmp_mcaptab) == Z_OK ||
2302                     zonecfg_get_aliased_rctl(handle, ALIAS_MAXSWAP, &tmp_mcap)
2303                     == Z_OK ||
2304                     zonecfg_get_aliased_rctl(handle, ALIAS_MAXLOCKEDMEM,
2305                     &tmp_mcap) == Z_OK) {
2306                         zerr(gettext("The %s resource or a related resource "
2307                             "control already exists."), rt_to_str(RT_MCAP));
2308                         goto bad;
2309                 }
2310                 if (global_zone)
2311                         zerr(gettext("WARNING: Setting a global zone memory "
2312                             "cap too low could deny\nservice "
2313                             "to even the root user; "
2314                             "this could render the system impossible\n"
2315                             "to administer.  Please use caution."));
2316                 bzero(&in_progress_mcaptab, sizeof (in_progress_mcaptab));
2317                 return;
2318         case RT_ADMIN:
2319                 bzero(&in_progress_admintab, sizeof (in_progress_admintab));
2320                 return;
2321         case RT_SECFLAGS:
2322                 /* Make sure we haven't already set this */
2323                 if (zonecfg_lookup_secflags(handle, &tmp_secflagstab) == Z_OK)
2324                         zerr(gettext("The %s resource already exists."),
2325                             rt_to_str(RT_SECFLAGS));
2326                 bzero(&in_progress_secflagstab,
2327                     sizeof (in_progress_secflagstab));
2328                 return;
2329         default:
2330                 zone_perror(rt_to_str(type), Z_NO_RESOURCE_TYPE, B_TRUE);
2331                 long_usage(CMD_ADD, B_TRUE);
2332                 usage(B_FALSE, HELP_RESOURCES);
2333         }
2334 bad:
2335         global_scope = B_TRUE;
2336         end_op = -1;
2337 }

2339 static void
2340 do_complex_rctl_val(complex_property_ptr_t cp)
2341 {
2342         struct zone_rctlvaltab *rctlvaltab;
2343         complex_property_ptr_t cx;
2344         boolean_t seen_priv = B_FALSE, seen_limit = B_FALSE,
2345             seen_action = B_FALSE;
2346         rctlblk_t *rctlblk;
2347         int err;

2349         if ((rctlvaltab = alloc_rctlvaltab()) == NULL) {
```

```
2350                 zone_perror(zone, Z_NOMEM, B_TRUE);
2351                 exit(Z_ERR);
2352         }
2353         for (cx = cp; cx != NULL; cx = cx->cp_next) {
2354                 switch (cx->cp_type) {
2355                 case PT_PRIV:
2356                         if (seen_priv) {
2357                                 zerr(gettext("%s already specified"),
2358                                     pt_to_str(PT_PRIV));
2359                                 goto bad;
2360                         }
2361                         (void) strlcpy(rctlvaltab->zone_rctlval_priv,
2362                             cx->cp_value,
2363                             sizeof (rctlvaltab->zone_rctlval_priv));
2364                         seen_priv = B_TRUE;
2365                         break;
2366                 case PT_LIMIT:
2367                         if (seen_limit) {
2368                                 zerr(gettext("%s already specified"),
2369                                     pt_to_str(PT_LIMIT));
2370                                 goto bad;
2371                         }
2372                         (void) strlcpy(rctlvaltab->zone_rctlval_limit,
2373                             cx->cp_value,
2374                             sizeof (rctlvaltab->zone_rctlval_limit));
2375                         seen_limit = B_TRUE;
2376                         break;
2377                 case PT_ACTION:
2378                         if (seen_action) {
2379                                 zerr(gettext("%s already specified"),
2380                                     pt_to_str(PT_ACTION));
2381                                 goto bad;
2382                         }
2383                         (void) strlcpy(rctlvaltab->zone_rctlval_action,
2384                             cx->cp_value,
2385                             sizeof (rctlvaltab->zone_rctlval_action));
2386                         seen_action = B_TRUE;
2387                         break;
2388                 default:
2389                         zone_perror(pt_to_str(PT_VALUE),
2390                             Z_NO_PROPERTY_TYPE, B_TRUE);
2391                         long_usage(CMD_ADD, B_TRUE);
2392                         usage(B_FALSE, HELP_PROPS);
2393                         zonecfg_free_rctl_value_list(rctlvaltab);
2394                         return;
2395                 }
2396         }
2397         if (!seen_priv)
2398                 zerr(gettext("%s not specified"), pt_to_str(PT_PRIV));
2399         if (!seen_limit)
2400                 zerr(gettext("%s not specified"), pt_to_str(PT_LIMIT));
2401         if (!seen_action)
2402                 zerr(gettext("%s not specified"), pt_to_str(PT_ACTION));
2403         if (!seen_priv || !seen_limit || !seen_action)
2404                 goto bad;
2405         rctlvaltab->zone_rctlval_next = NULL;
2406         rctlblk = alloca(rctlblk_size());
2407         /*
2408          * Make sure the rctl value looks roughly correct; we won't know if
2409          * it's truly OK until we verify the configuration on the target
2410          * system.
2411          */
2412         if (zonecfg_construct_rctlblk(rctlvaltab, rctlblk) != Z_OK ||
2413             !zonecfg_valid_rctlblk(rctlblk)) {
2414                 zerr(gettext("Invalid %s %s specification"), rt_to_str(RT_RCTL),
2415                     pt_to_str(PT_VALUE));
```



```
2416                 goto bad;
2417         }
2418         err = zonecfg_add_rctl_value(&in_progress_rctltab, rctlvaltab);
2419         if (err != Z_OK)
2420                 zone_perror(pt_to_str(PT_VALUE), err, B_TRUE);
2421         return;
2422
2423 bad:
2424         zonecfg_free_rctl_value_list(rctlvaltab);
2425 }
2426
2427 static void
2428 add_property(cmd_t *cmd)
2429 {
2430         char *prop_id;
2431         int err, res_type, prop_type;
2432         property_value_ptr_t pp;
2433         list_property_ptr_t l;
2434
2435         res_type = resource_scope;
2436         prop_type = cmd->cmd_prop_name[0];
2437         if (res_type == RT_UNKNOWN || prop_type == PT_UNKNOWN) {
2438                 long_usage(CMD_ADD, B_TRUE);
2439                 return;
2440         }
2441
2442         if (cmd->cmd_prop_nv_pairs != 1) {
2443                 long_usage(CMD_ADD, B_TRUE);
2444                 return;
2445         }
2446
2447         if (initialize(B_TRUE) != Z_OK)
2448                 return;
2449
2450         switch (res_type) {
2451         case RT_FS:
2452                 if (prop_type != PT_OPTIONS) {
2453                         zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
2454                             B_TRUE);
2455                         long_usage(CMD_ADD, B_TRUE);
2456                         usage(B_FALSE, HELP_PROPS);
2457                         return;
2458                 }
2459                 pp = cmd->cmd_property_ptr[0];
2460                 if (pp->pv_type != PROP_VAL_SIMPLE &&
2461                     pp->pv_type != PROP_VAL_LIST) {
2462                         zerr(gettext("A %s or %s value was expected here."),
2463                             pvt_to_str(PROP_VAL_SIMPLE),
2464                             pvt_to_str(PROP_VAL_LIST));
2465                         saw_error = B_TRUE;
2466                         return;
2467                 }
2468                 if (pp->pv_type == PROP_VAL_SIMPLE) {
2469                         if (pp->pv_simple == NULL) {
2470                                 long_usage(CMD_ADD, B_TRUE);
2471                                 return;
2472                         }
2473                         prop_id = pp->pv_simple;
2474                         err = zonecfg_add_fs_option(&in_progress_fstab,
2475                             prop_id);
2476                         if (err != Z_OK)
2477                                 zone_perror(pt_to_str(prop_type), err, B_TRUE);
2478                 } else {
2479                         list_property_ptr_t list;
2480
2481                         for (list = pp->pv_list; list != NULL;
```

```
2482                            list = list->lp_next) {
2483                                prop_id = list->lp_simple;
2484                                if (prop_id == NULL)
2485                                        break;
2486                                err = zonecfg_add_fs_option(
2487                                    &in_progress_fstab, prop_id);
2488                                if (err != Z_OK)
2489                                        zone_perror(pt_to_str(prop_type), err,
2490                                            B_TRUE);
2491                        }
2492                }
2493                return;
2494        case RT_RCTL:
2495                if (prop_type != PT_VALUE) {
2496                        zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
2497                            B_TRUE);
2498                        long_usage(CMD_ADD, B_TRUE);
2499                        usage(B_FALSE, HELP_PROPS);
2500                        return;
2501                }
2502                pp = cmd->cmd_property_ptr[0];
2503                if (pp->pv_type != PROP_VAL_COMPLEX &&
2504                    pp->pv_type != PROP_VAL_LIST) {
2505                        zerr(gettext("A %s or %s value was expected here."),
2506                            pvt_to_str(PROP_VAL_COMPLEX),
2507                            pvt_to_str(PROP_VAL_LIST));
2508                        saw_error = B_TRUE;
2509                        return;
2510                }
2511                if (pp->pv_type == PROP_VAL_COMPLEX) {
2512                        do_complex_rctl_val(pp->pv_complex);
2513                        return;
2514                }
2515                for (l = pp->pv_list; l != NULL; l = l->lp_next)
2516                        do_complex_rctl_val(l->lp_complex);
2517                return;
2518        default:
2519                zone_perror(rt_to_str(res_type), Z_NO_RESOURCE_TYPE, B_TRUE);
2520                long_usage(CMD_ADD, B_TRUE);
2521                usage(B_FALSE, HELP_RESOURCES);
2522                return;
2523        }
2524 }

2526 static boolean_t
2527 gz_invalid_resource(int type)
2528 {
2529        return (global_zone && (type == RT_FS ||
2530            type == RT_NET || type == RT_DEVICE || type == RT_ATTR ||
2531            type == RT_DATASET));
2532 }

2534 static boolean_t
2535 gz_invalid_rt_property(int type)
2536 {
2537        return (global_zone && (type == RT_ZONENAME || type == RT_ZONEPATH ||
2538            type == RT_AUTOBOOT || type == RT_LIMITPRIV ||
2539            type == RT_BOOTARGS || type == RT_BRAND || type == RT_SCHED ||
2540            type == RT_IPTYPE || type == RT_HOSTID || type == RT_FS_ALLOWED));
2541 }

2543 static boolean_t
2544 gz_invalid_property(int type)
2545 {
2546        return (global_zone && (type == PT_ZONENAME || type == PT_ZONEPATH ||
2547            type == PT_AUTOBOOT || type == PT_LIMITPRIV ||
```



```
2548            type == PT_BOOTARGS || type == PT_BRAND || type == PT_SCHED ||
2549            type == PT_IPTYPE || type == PT_HOSTID || type == PT_FS_ALLOWED));
2550 }

2552 void
2553 add_func(cmd_t *cmd)
2554 {
2555        int arg;
2556        boolean_t arg_err = B_FALSE;

2558        assert(cmd != NULL);

2560        optind = 0;
2561        while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "?")) != EOF) {
2562                switch (arg) {
2563                case '?':
2564                        longer_usage(CMD_ADD);
2565                        arg_err = B_TRUE;
2566                        break;
2567                default:
2568                        short_usage(CMD_ADD);
2569                        arg_err = B_TRUE;
2570                        break;
2571                }
2572        }
2573        if (arg_err)
2574                return;

2576        if (optind != cmd->cmd_argc) {
2577                short_usage(CMD_ADD);
2578                return;
2579        }

2581        if (zone_is_read_only(CMD_ADD))
2582                return;

2584        if (initialize(B_TRUE) != Z_OK)
2585                return;
2586        if (global_scope) {
2587                if (gz_invalid_resource(cmd->cmd_res_type)) {
2588                        zerr(gettext("Cannot add a %s resource to the "
2589                            "global zone."), rt_to_str(cmd->cmd_res_type));
2590                        saw_error = B_TRUE;
2591                        return;
2592                }

2594                global_scope = B_FALSE;
2595                resource_scope = cmd->cmd_res_type;
2596                end_op = CMD_ADD;
2597                add_resource(cmd);
2598        } else
2599                add_property(cmd);
2600 }

2602 /*
2603  * This routine has an unusual implementation, because it tries very
2604  * hard to succeed in the face of a variety of failure modes.
2605  * The most common and most vexing occurs when the index file and
2606  * the /etc/zones/<zonename.xml> file are not both present.  In
2607  * this case, delete must eradicate as much of the zone state as is left
2608  * so that the user can later create a new zone with the same name.
2609  */
2610 void
2611 delete_func(cmd_t *cmd)
2612 {
2613        int err, arg, answer;
```

```
2614         char line[ZONENAME_MAX + 128];  /* enough to ask a question */
2615         boolean_t force = B_FALSE;
2616         boolean_t arg_err = B_FALSE;

2618         optind = 0;
2619         while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "?F")) != EOF) {
2620                 switch (arg) {
2621                 case '?':
2622                         longer_usage(CMD_DELETE);
2623                         arg_err = B_TRUE;
2624                         break;
2625                 case 'F':
2626                         force = B_TRUE;
2627                         break;
2628                 default:
2629                         short_usage(CMD_DELETE);
2630                         arg_err = B_TRUE;
2631                         break;
2632                 }
2633         }
2634         if (arg_err)
2635                 return;

2637         if (optind != cmd->cmd_argc) {
2638                 short_usage(CMD_DELETE);
2639                 return;
2640         }

2642         if (zone_is_read_only(CMD_DELETE))
2643                 return;

2645         if (!force) {
2646                 /*
2647                  * Initialize sets up the global called "handle" and warns the
2648                  * user if the zone is not configured.  In force mode, we don't
2649                  * trust that evaluation, and hence skip it.  (We don't need the
2650                  * handle to be loaded anyway, since zonecfg_destroy is done by
2651                  * zonename). However, we also have to take care to emulate the
2652                  * messages spit out by initialize; see below.
2653                  */
2654                 if (initialize(B_TRUE) != Z_OK)
2655                         return;

2657                 (void) snprintf(line, sizeof (line),
2658                     gettext("Are you sure you want to delete zone %s"), zone);
2659                 if ((answer = ask_yesno(B_FALSE, line)) == -1) {
2660                         zerr(gettext("Input not from terminal and -F not "
2661                             "specified:\n%s command ignored, exiting."),
2662                             cmd_to_str(CMD_DELETE));
2663                         exit(Z_ERR);
2664                 }
2665                 if (answer != 1)
2666                         return;
2667         }

2669         /*
2670          * This function removes the authorizations from user_attr
2671          * that correspond to those specified in the configuration
2672          */
2673         if (initialize(B_TRUE) == Z_OK) {
2674                 (void) zonecfg_deauthorize_users(handle, zone);
2675         }
2676         if ((err = zonecfg_destroy(zone, force)) != Z_OK) {
2677                 if ((err == Z_BAD_ZONE_STATE) && !force) {
2678                         zerr(gettext("Zone %s not in %s state; %s not "
2679                             "allowed.  Use -F to force %s."),
```



```
2680                             zone, zone_state_str(ZONE_STATE_CONFIGURED),
2681                             cmd_to_str(CMD_DELETE), cmd_to_str(CMD_DELETE));
2682                 } else {
2683                         zone_perror(zone, err, B_TRUE);
2684                 }
2685         }
2686         need_to_commit = B_FALSE;

2688         /*
2689          * Emulate initialize's messaging; if there wasn't a valid handle to
2690          * begin with, then user had typed delete (or delete -F) multiple
2691          * times.  So we emit a message.
2692          *
2693          * We only do this in the 'force' case because normally, initialize()
2694          * takes care of this for us.
2695          */
2696         if (force && zonecfg_check_handle(handle) != Z_OK && interactive_mode)
2697                 (void) printf(gettext("Use '%s' to begin "
2698                     "configuring a new zone.\n"), cmd_to_str(CMD_CREATE));

2700         /*
2701          * Time for a new handle: finish the old one off first
2702          * then get a new one properly to avoid leaks.
2703          */
2704         if (got_handle) {
2705                 zonecfg_fini_handle(handle);
2706                 if ((handle = zonecfg_init_handle()) == NULL) {
2707                         zone_perror(execname, Z_NOMEM, B_TRUE);
2708                         exit(Z_ERR);
2709                 }
2710                 if ((err = zonecfg_get_handle(zone, handle)) != Z_OK) {
2711                         /* If there was no zone before, that's OK */
2712                         if (err != Z_NO_ZONE)
2713                                 zone_perror(zone, err, B_TRUE);
2714                         got_handle = B_FALSE;
2715                 }
2716         }
2717 }

2719 static int
2720 fill_in_fstab(cmd_t *cmd, struct zone_fstab *fstab, boolean_t fill_in_only)
2721 {
2722         int err, i;
2723         property_value_ptr_t pp;

2725         if ((err = initialize(B_TRUE)) != Z_OK)
2726                 return (err);

2728         bzero(fstab, sizeof (*fstab));
2729         for (i = 0; i < cmd->cmd_prop_nv_pairs; i++) {
2730                 pp = cmd->cmd_property_ptr[i];
2731                 if (pp->pv_type != PROP_VAL_SIMPLE || pp->pv_simple == NULL) {
2732                         zerr(gettext("A simple value was expected here."));
2733                         saw_error = B_TRUE;
2734                         return (Z_INSUFFICIENT_SPEC);
2735                 }
2736                 switch (cmd->cmd_prop_name[i]) {
2737                 case PT_DIR:
2738                         (void) strlcpy(fstab->zone_fs_dir, pp->pv_simple,
2739                             sizeof (fstab->zone_fs_dir));
2740                         break;
2741                 case PT_SPECIAL:
2742                         (void) strlcpy(fstab->zone_fs_special, pp->pv_simple,
2743                             sizeof (fstab->zone_fs_special));
2744                         break;
2745                 case PT_RAW:
```

```
2746                        (void) strlcpy(fstab->zone_fs_raw, pp->pv_simple,
2747                            sizeof (fstab->zone_fs_raw));
2748                        break;
2749                case PT_TYPE:
2750                        (void) strlcpy(fstab->zone_fs_type, pp->pv_simple,
2751                            sizeof (fstab->zone_fs_type));
2752                        break;
2753                default:
2754                        zone_perror(pt_to_str(cmd->cmd_prop_name[i]),
2755                            Z_NO_PROPERTY_TYPE, B_TRUE);
2756                        return (Z_INSUFFICIENT_SPEC);
2757                }
2758        }
2759        if (fill_in_only)
2760                return (Z_OK);
2761        return (zonecfg_lookup_filesystem(handle, fstab));
2762 }

2764 static int
2765 fill_in_nwiftab(cmd_t *cmd, struct zone_nwiftab *nwiftab,
2766     boolean_t fill_in_only)
2767 {
2768        int err, i;
2769        property_value_ptr_t pp;

2771        if ((err = initialize(B_TRUE)) != Z_OK)
2772                return (err);

2774        bzero(nwiftab, sizeof (*nwiftab));
2775        for (i = 0; i < cmd->cmd_prop_nv_pairs; i++) {
2776                pp = cmd->cmd_property_ptr[i];
2777                if (pp->pv_type != PROP_VAL_SIMPLE || pp->pv_simple == NULL) {
2778                        zerr(gettext("A simple value was expected here."));
2779                        saw_error = B_TRUE;
2780                        return (Z_INSUFFICIENT_SPEC);
2781                }
2782                switch (cmd->cmd_prop_name[i]) {
2783                case PT_ADDRESS:
2784                        (void) strlcpy(nwiftab->zone_nwif_address,
2785                            pp->pv_simple, sizeof (nwiftab->zone_nwif_address));
2786                        break;
2787                case PT_ALLOWED_ADDRESS:
2788                        (void) strlcpy(nwiftab->zone_nwif_allowed_address,
2789                            pp->pv_simple,
2790                            sizeof (nwiftab->zone_nwif_allowed_address));
2791                        break;
2792                case PT_PHYSICAL:
2793                        (void) strlcpy(nwiftab->zone_nwif_physical,
2794                            pp->pv_simple,
2795                            sizeof (nwiftab->zone_nwif_physical));
2796                        break;
2797                case PT_DEFROUTER:
2798                        (void) strlcpy(nwiftab->zone_nwif_defrouter,
2799                            pp->pv_simple,
2800                            sizeof (nwiftab->zone_nwif_defrouter));
2801                        break;
2802                default:
2803                        zone_perror(pt_to_str(cmd->cmd_prop_name[i]),
2804                            Z_NO_PROPERTY_TYPE, B_TRUE);
2805                        return (Z_INSUFFICIENT_SPEC);
2806                }
2807        }
2808        if (fill_in_only)
2809                return (Z_OK);
2810        err = zonecfg_lookup_nwif(handle, nwiftab);
2811        return (err);
```



```
2812 }

2814 static int
2815 fill_in_devtab(cmd_t *cmd, struct zone_devtab *devtab, boolean_t fill_in_only)
2816 {
2817        int err, i;
2818        property_value_ptr_t pp;

2820        if ((err = initialize(B_TRUE)) != Z_OK)
2821                return (err);

2823        bzero(devtab, sizeof (*devtab));
2824        for (i = 0; i < cmd->cmd_prop_nv_pairs; i++) {
2825                pp = cmd->cmd_property_ptr[i];
2826                if (pp->pv_type != PROP_VAL_SIMPLE || pp->pv_simple == NULL) {
2827                        zerr(gettext("A simple value was expected here."));
2828                        saw_error = B_TRUE;
2829                        return (Z_INSUFFICIENT_SPEC);
2830                }
2831                switch (cmd->cmd_prop_name[i]) {
2832                case PT_MATCH:
2833                        (void) strlcpy(devtab->zone_dev_match, pp->pv_simple,
2834                            sizeof (devtab->zone_dev_match));
2835                        break;
2836                default:
2837                        zone_perror(pt_to_str(cmd->cmd_prop_name[i]),
2838                            Z_NO_PROPERTY_TYPE, B_TRUE);
2839                        return (Z_INSUFFICIENT_SPEC);
2840                }
2841        }
2842        if (fill_in_only)
2843                return (Z_OK);
2844        err = zonecfg_lookup_dev(handle, devtab);
2845        return (err);
2846 }

2848 static int
2849 fill_in_rctltab(cmd_t *cmd, struct zone_rctltab *rctltab,
2850     boolean_t fill_in_only)
2851 {
2852        int err, i;
2853        property_value_ptr_t pp;

2855        if ((err = initialize(B_TRUE)) != Z_OK)
2856                return (err);

2858        bzero(rctltab, sizeof (*rctltab));
2859        for (i = 0; i < cmd->cmd_prop_nv_pairs; i++) {
2860                pp = cmd->cmd_property_ptr[i];
2861                if (pp->pv_type != PROP_VAL_SIMPLE || pp->pv_simple == NULL) {
2862                        zerr(gettext("A simple value was expected here."));
2863                        saw_error = B_TRUE;
2864                        return (Z_INSUFFICIENT_SPEC);
2865                }
2866                switch (cmd->cmd_prop_name[i]) {
2867                case PT_NAME:
2868                        (void) strlcpy(rctltab->zone_rctl_name, pp->pv_simple,
2869                            sizeof (rctltab->zone_rctl_name));
2870                        break;
2871                default:
2872                        zone_perror(pt_to_str(cmd->cmd_prop_name[i]),
2873                            Z_NO_PROPERTY_TYPE, B_TRUE);
2874                        return (Z_INSUFFICIENT_SPEC);
2875                }
2876        }
2877        if (fill_in_only)
```

```
2878                 return (Z_OK);
2879         err = zonecfg_lookup_rctl(handle, rctltab);
2880         return (err);
2881 }

2883 static int
2884 fill_in_attrtab(cmd_t *cmd, struct zone_attrtab *attrtab,
2885     boolean_t fill_in_only)
2886 {
2887         int err, i;
2888         property_value_ptr_t pp;

2890         if ((err = initialize(B_TRUE)) != Z_OK)
2891                 return (err);

2893         bzero(attrtab, sizeof (*attrtab));
2894         for (i = 0; i < cmd->cmd_prop_nv_pairs; i++) {
2895                 pp = cmd->cmd_property_ptr[i];
2896                 if (pp->pv_type != PROP_VAL_SIMPLE || pp->pv_simple == NULL) {
2897                         zerr(gettext("A simple value was expected here."));
2898                         saw_error = B_TRUE;
2899                         return (Z_INSUFFICIENT_SPEC);
2900                 }
2901                 switch (cmd->cmd_prop_name[i]) {
2902                 case PT_NAME:
2903                         (void) strlcpy(attrtab->zone_attr_name, pp->pv_simple,
2904                             sizeof (attrtab->zone_attr_name));
2905                         break;
2906                 case PT_TYPE:
2907                         (void) strlcpy(attrtab->zone_attr_type, pp->pv_simple,
2908                             sizeof (attrtab->zone_attr_type));
2909                         break;
2910                 case PT_VALUE:
2911                         (void) strlcpy(attrtab->zone_attr_value, pp->pv_simple,
2912                             sizeof (attrtab->zone_attr_value));
2913                         break;
2914                 default:
2915                         zone_perror(pt_to_str(cmd->cmd_prop_name[i]),
2916                             Z_NO_PROPERTY_TYPE, B_TRUE);
2917                         return (Z_INSUFFICIENT_SPEC);
2918                 }
2919         }
2920         if (fill_in_only)
2921                 return (Z_OK);
2922         err = zonecfg_lookup_attr(handle, attrtab);
2923         return (err);
2924 }

2926 static int
2927 fill_in_dstab(cmd_t *cmd, struct zone_dstab *dstab, boolean_t fill_in_only)
2928 {
2929         int err, i;
2930         property_value_ptr_t pp;

2932         if ((err = initialize(B_TRUE)) != Z_OK)
2933                 return (err);

2935         dstab->zone_dataset_name[0] = '\0';
2936         for (i = 0; i < cmd->cmd_prop_nv_pairs; i++) {
2937                 pp = cmd->cmd_property_ptr[i];
2938                 if (pp->pv_type != PROP_VAL_SIMPLE || pp->pv_simple == NULL) {
2939                         zerr(gettext("A simple value was expected here."));
2940                         saw_error = B_TRUE;
2941                         return (Z_INSUFFICIENT_SPEC);
2942                 }
2943                 switch (cmd->cmd_prop_name[i]) {
```



```
2944                 case PT_NAME:
2945                         (void) strlcpy(dstab->zone_dataset_name, pp->pv_simple,
2946                             sizeof (dstab->zone_dataset_name));
2947                         break;
2948                 default:
2949                         zone_perror(pt_to_str(cmd->cmd_prop_name[i]),
2950                             Z_NO_PROPERTY_TYPE, B_TRUE);
2951                         return (Z_INSUFFICIENT_SPEC);
2952                 }
2953         }
2954         if (fill_in_only)
2955                 return (Z_OK);
2956         return (zonecfg_lookup_ds(handle, dstab));
2957 }

2959 static int
2960 fill_in_admintab(cmd_t *cmd, struct zone_admintab *admintab,
2961     boolean_t fill_in_only)
2962 {
2963         int err, i;
2964         property_value_ptr_t pp;

2966         if ((err = initialize(B_TRUE)) != Z_OK)
2967                 return (err);

2969         bzero(admintab, sizeof (*admintab));
2970         for (i = 0; i < cmd->cmd_prop_nv_pairs; i++) {
2971                 pp = cmd->cmd_property_ptr[i];
2972                 if (pp->pv_type != PROP_VAL_SIMPLE || pp->pv_simple == NULL) {
2973                         zerr(gettext("A simple value was expected here."));
2974                         saw_error = B_TRUE;
2975                         return (Z_INSUFFICIENT_SPEC);
2976                 }
2977                 switch (cmd->cmd_prop_name[i]) {
2978                 case PT_USER:
2979                         (void) strlcpy(admintab->zone_admin_user, pp->pv_simple,
2980                             sizeof (admintab->zone_admin_user));
2981                         break;
2982                 case PT_AUTHS:
2983                         (void) strlcpy(admintab->zone_admin_auths,
2984                             pp->pv_simple, sizeof (admintab->zone_admin_auths));
2985                         break;
2986                 default:
2987                         zone_perror(pt_to_str(cmd->cmd_prop_name[i]),
2988                             Z_NO_PROPERTY_TYPE, B_TRUE);
2989                         return (Z_INSUFFICIENT_SPEC);
2990                 }
2991         }
2992         if (fill_in_only)
2993                 return (Z_OK);
2994         err = zonecfg_lookup_admin(handle, admintab);
2995         return (err);
2996 }

2998 static int
2999 fill_in_secflagstab(cmd_t *cmd, struct zone_secflagstab *secflagstab,
3000     boolean_t fill_in_only)
3001 {
3002         int err, i;
3003         property_value_ptr_t pp;

3005         if ((err = initialize(B_TRUE)) != Z_OK)
3006                 return (err);

3008         bzero(secflagstab, sizeof (*secflagstab));
3009         for (i = 0; i < cmd->cmd_prop_nv_pairs; i++) {
```

```
3010                 pp = cmd->cmd_property_ptr[i];
3011                 if (pp->pv_type != PROP_VAL_SIMPLE || pp->pv_simple == NULL) {
3012                         zerr(gettext("A simple value was expected here."));
3013                         saw_error = B_TRUE;
3014                         return (Z_INSUFFICIENT_SPEC);
3015                 }
3016                 switch (cmd->cmd_prop_name[i]) {
3017                 case PT_DEFAULT:
3018                         (void) strlcpy(secflagstab->zone_secflags_default,
3019                             pp->pv_simple,
3020                             sizeof (secflagstab->zone_secflags_default));
3021                         break;
3022                 case PT_LOWER:
3023                         (void) strlcpy(secflagstab->zone_secflags_lower,
3024                             pp->pv_simple,
3025                             sizeof (secflagstab->zone_secflags_lower));
3026                         break;
3027                 case PT_UPPER:
3028                         (void) strlcpy(secflagstab->zone_secflags_upper,
3029                             pp->pv_simple,
3030                             sizeof (secflagstab->zone_secflags_upper));
3031                         break;
3032                 default:
3033                         zone_perror(pt_to_str(cmd->cmd_prop_name[i]),
3034                             Z_NO_PROPERTY_TYPE, B_TRUE);
3035                         return (Z_INSUFFICIENT_SPEC);
3036                 }
3037         }
3038         if (fill_in_only)
3039                 return (Z_OK);

3041         err = zonecfg_lookup_secflags(handle, secflagstab);

3043         return (err);
3044 }

3046 static void
3047 remove_aliased_rctl(int type, char *name)
3048 {
3049         int err;
3050         uint64_t tmp;

3052         if ((err = zonecfg_get_aliased_rctl(handle, name, &tmp)) != Z_OK) {
3053                 zerr("%s %s: %s", cmd_to_str(CMD_CLEAR), pt_to_str(type),
3054                     zonecfg_strerror(err));
3055                 saw_error = B_TRUE;
3056                 return;
3057         }
3058         if ((err = zonecfg_rm_aliased_rctl(handle, name)) != Z_OK) {
3059                 zerr("%s %s: %s", cmd_to_str(CMD_CLEAR), pt_to_str(type),
3060                     zonecfg_strerror(err));
3061                 saw_error = B_TRUE;
3062         } else {
3063                 need_to_commit = B_TRUE;
3064         }
3065 }

3067 static boolean_t
3068 prompt_remove_resource(cmd_t *cmd, char *rsrc)
3069 {
3070         int num;
3071         int answer;
3072         int arg;
3073         boolean_t force = B_FALSE;
3074         char prompt[128];
3075         boolean_t arg_err = B_FALSE;
```



```
3077         optind = 0;
3078         while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "F")) != EOF) {
3079                 switch (arg) {
3080                 case 'F':
3081                         force = B_TRUE;
3082                         break;
3083                 default:
3084                         arg_err = B_TRUE;
3085                         break;
3086                 }
3087         }
3088         if (arg_err)
3089                 return (B_FALSE);


3092         num = zonecfg_num_resources(handle, rsrc);

3094         if (num == 0) {
3095                 z_cmd_rt_perror(CMD_REMOVE, cmd->cmd_res_type, Z_NO_ENTRY,
3096                     B_TRUE);
3097                 return (B_FALSE);
3098         }
3099         if (num > 1 && !force) {
3100                 if (!interactive_mode) {
3101                         zerr(gettext("There are multiple instances of this "
3102                             "resource.  Either qualify the resource to\n"
3103                             "remove a single instance or use the -F option to "
3104                             "remove all instances."));
3105                         saw_error = B_TRUE;
3106                         return (B_FALSE);
3107                 }
3108                 (void) snprintf(prompt, sizeof (prompt), gettext(
3109                     "Are you sure you want to remove ALL '%s' resources"),
3110                     rsrc);
3111                 answer = ask_yesno(B_FALSE, prompt);
3112                 if (answer == -1) {
3113                         zerr(gettext("Resource incomplete."));
3114                         return (B_FALSE);
3115                 }
3116                 if (answer != 1)
3117                         return (B_FALSE);
3118         }
3119         return (B_TRUE);
3120 }

3122 static void
3123 remove_fs(cmd_t *cmd)
3124 {
3125         int err;

3127         /* traditional, qualified fs removal */
3128         if (cmd->cmd_prop_nv_pairs > 0) {
3129                 struct zone_fstab fstab;

3131                 if ((err = fill_in_fstab(cmd, &fstab, B_FALSE)) != Z_OK) {
3132                         z_cmd_rt_perror(CMD_REMOVE, RT_FS, err, B_TRUE);
3133                         return;
3134                 }
3135                 if ((err = zonecfg_delete_filesystem(handle, &fstab)) != Z_OK)
3136                         z_cmd_rt_perror(CMD_REMOVE, RT_FS, err, B_TRUE);
3137                 else
3138                         need_to_commit = B_TRUE;
3139                 zonecfg_free_fs_option_list(fstab.zone_fs_options);
3140                 return;
3141         }
```

```
3143         /*
3144          * unqualified fs removal.  remove all fs's but prompt if more
3145          * than one.
3146          */
3147         if (!prompt_remove_resource(cmd, "fs"))
3148                 return;

3150         if ((err = zonecfg_del_all_resources(handle, "fs")) != Z_OK)
3151                 z_cmd_rt_perror(CMD_REMOVE, RT_FS, err, B_TRUE);
3152         else
3153                 need_to_commit = B_TRUE;
3154 }

3156 static void
3157 remove_net(cmd_t *cmd)
3158 {
3159         int err;

3161         /* traditional, qualified net removal */
3162         if (cmd->cmd_prop_nv_pairs > 0) {
3163                 struct zone_nwiftab nwiftab;

3165                 if ((err = fill_in_nwiftab(cmd, &nwiftab, B_FALSE)) != Z_OK) {
3166                         z_cmd_rt_perror(CMD_REMOVE, RT_NET, err, B_TRUE);
3167                         return;
3168                 }
3169                 if ((err = zonecfg_delete_nwif(handle, &nwiftab)) != Z_OK)
3170                         z_cmd_rt_perror(CMD_REMOVE, RT_NET, err, B_TRUE);
3171                 else
3172                         need_to_commit = B_TRUE;
3173                 return;
3174         }

3176         /*
3177          * unqualified net removal.  remove all nets but prompt if more
3178          * than one.
3179          */
3180         if (!prompt_remove_resource(cmd, "net"))
3181                 return;

3183         if ((err = zonecfg_del_all_resources(handle, "net")) != Z_OK)
3184                 z_cmd_rt_perror(CMD_REMOVE, RT_NET, err, B_TRUE);
3185         else
3186                 need_to_commit = B_TRUE;
3187 }

3189 static void
3190 remove_device(cmd_t *cmd)
3191 {
3192         int err;

3194         /* traditional, qualified device removal */
3195         if (cmd->cmd_prop_nv_pairs > 0) {
3196                 struct zone_devtab devtab;

3198                 if ((err = fill_in_devtab(cmd, &devtab, B_FALSE)) != Z_OK) {
3199                         z_cmd_rt_perror(CMD_REMOVE, RT_DEVICE, err, B_TRUE);
3200                         return;
3201                 }
3202                 if ((err = zonecfg_delete_dev(handle, &devtab)) != Z_OK)
3203                         z_cmd_rt_perror(CMD_REMOVE, RT_DEVICE, err, B_TRUE);
3204                 else
3205                         need_to_commit = B_TRUE;
3206                 return;
3207         }
```



```
3209         /*
3210          * unqualified device removal.  remove all devices but prompt if more
3211          * than one.
3212          */
3213         if (!prompt_remove_resource(cmd, "device"))
3214                 return;

3216         if ((err = zonecfg_del_all_resources(handle, "device")) != Z_OK)
3217                 z_cmd_rt_perror(CMD_REMOVE, RT_DEVICE, err, B_TRUE);
3218         else
3219                 need_to_commit = B_TRUE;
3220 }

3222 static void
3223 remove_attr(cmd_t *cmd)
3224 {
3225         int err;

3227         /* traditional, qualified attr removal */
3228         if (cmd->cmd_prop_nv_pairs > 0) {
3229                 struct zone_attrtab attrtab;

3231                 if ((err = fill_in_attrtab(cmd, &attrtab, B_FALSE)) != Z_OK) {
3232                         z_cmd_rt_perror(CMD_REMOVE, RT_ATTR, err, B_TRUE);
3233                         return;
3234                 }
3235                 if ((err = zonecfg_delete_attr(handle, &attrtab)) != Z_OK)
3236                         z_cmd_rt_perror(CMD_REMOVE, RT_ATTR, err, B_TRUE);
3237                 else
3238                         need_to_commit = B_TRUE;
3239                 return;
3240         }

3242         /*
3243          * unqualified attr removal.  remove all attrs but prompt if more
3244          * than one.
3245          */
3246         if (!prompt_remove_resource(cmd, "attr"))
3247                 return;

3249         if ((err = zonecfg_del_all_resources(handle, "attr")) != Z_OK)
3250                 z_cmd_rt_perror(CMD_REMOVE, RT_ATTR, err, B_TRUE);
3251         else
3252                 need_to_commit = B_TRUE;
3253 }

3255 static void
3256 remove_dataset(cmd_t *cmd)
3257 {
3258         int err;

3260         /* traditional, qualified dataset removal */
3261         if (cmd->cmd_prop_nv_pairs > 0) {
3262                 struct zone_dstab dstab;

3264                 if ((err = fill_in_dstab(cmd, &dstab, B_FALSE)) != Z_OK) {
3265                         z_cmd_rt_perror(CMD_REMOVE, RT_DATASET, err, B_TRUE);
3266                         return;
3267                 }
3268                 if ((err = zonecfg_delete_ds(handle, &dstab)) != Z_OK)
3269                         z_cmd_rt_perror(CMD_REMOVE, RT_DATASET, err, B_TRUE);
3270                 else
3271                         need_to_commit = B_TRUE;
3272                 return;
3273         }
```

```
3275         /*
3276          * unqualified dataset removal.  remove all datasets but prompt if more
3277          * than one.
3278          */
3279         if (!prompt_remove_resource(cmd, "dataset"))
3280                 return;

3282         if ((err = zonecfg_del_all_resources(handle, "dataset")) != Z_OK)
3283                 z_cmd_rt_perror(CMD_REMOVE, RT_DATASET, err, B_TRUE);
3284         else
3285                 need_to_commit = B_TRUE;
3286 }

3288 static void
3289 remove_rctl(cmd_t *cmd)
3290 {
3291         int err;

3293         /* traditional, qualified rctl removal */
3294         if (cmd->cmd_prop_nv_pairs > 0) {
3295                 struct zone_rctltab rctltab;

3297                 if ((err = fill_in_rctltab(cmd, &rctltab, B_FALSE)) != Z_OK) {
3298                         z_cmd_rt_perror(CMD_REMOVE, RT_RCTL, err, B_TRUE);
3299                         return;
3300                 }
3301                 if ((err = zonecfg_delete_rctl(handle, &rctltab)) != Z_OK)
3302                         z_cmd_rt_perror(CMD_REMOVE, RT_RCTL, err, B_TRUE);
3303                 else
3304                         need_to_commit = B_TRUE;
3305                 zonecfg_free_rctl_value_list(rctltab.zone_rctl_valptr);
3306                 return;
3307         }

3309         /*
3310          * unqualified rctl removal.  remove all rctls but prompt if more
3311          * than one.
3312          */
3313         if (!prompt_remove_resource(cmd, "rctl"))
3314                 return;

3316         if ((err = zonecfg_del_all_resources(handle, "rctl")) != Z_OK)
3317                 z_cmd_rt_perror(CMD_REMOVE, RT_RCTL, err, B_TRUE);
3318         else
3319                 need_to_commit = B_TRUE;
3320 }

3322 static void
3323 remove_pset()
3324 {
3325         int err;
3326         struct zone_psettab psettab;

3328         if ((err = zonecfg_lookup_pset(handle, &psettab)) != Z_OK) {
3329                 z_cmd_rt_perror(CMD_REMOVE, RT_DCPU, err, B_TRUE);
3330                 return;
3331         }
3332         if ((err = zonecfg_delete_pset(handle)) != Z_OK)
3333                 z_cmd_rt_perror(CMD_REMOVE, RT_DCPU, err, B_TRUE);
3334         else
3335                 need_to_commit = B_TRUE;
3336 }

3338 static void
3339 remove_pcap()
```



```
3340 {
3341         int err;
3342         uint64_t tmp;

3344         if (zonecfg_get_aliased_rctl(handle, ALIAS_CPUCAP, &tmp) != Z_OK) {
3345                 zerr("%s %s: %s", cmd_to_str(CMD_REMOVE), rt_to_str(RT_PCAP),
3346                     zonecfg_strerror(Z_NO_RESOURCE_TYPE));
3347                 saw_error = B_TRUE;
3348                 return;
3349         }

3351         if ((err = zonecfg_rm_aliased_rctl(handle, ALIAS_CPUCAP)) != Z_OK)
3352                 z_cmd_rt_perror(CMD_REMOVE, RT_PCAP, err, B_TRUE);
3353         else
3354                 need_to_commit = B_TRUE;
3355 }

3357 static void
3358 remove_mcap()
3359 {
3360         int err, res1, res2, res3;
3361         uint64_t tmp;
3362         struct zone_mcaptab mcaptab;
3363         boolean_t revert = B_FALSE;

3365         res1 = zonecfg_lookup_mcap(handle, &mcaptab);
3366         res2 = zonecfg_get_aliased_rctl(handle, ALIAS_MAXSWAP, &tmp);
3367         res3 = zonecfg_get_aliased_rctl(handle, ALIAS_MAXLOCKEDMEM, &tmp);

3369         /* if none of these exist, there is no resource to remove */
3370         if (res1 != Z_OK && res2 != Z_OK && res3 != Z_OK) {
3371                 zerr("%s %s: %s", cmd_to_str(CMD_REMOVE), rt_to_str(RT_MCAP),
3372                     zonecfg_strerror(Z_NO_RESOURCE_TYPE));
3373                 saw_error = B_TRUE;
3374                 return;
3375         }
3376         if (res1 == Z_OK) {
3377                 if ((err = zonecfg_delete_mcap(handle)) != Z_OK) {
3378                         z_cmd_rt_perror(CMD_REMOVE, RT_MCAP, err, B_TRUE);
3379                         revert = B_TRUE;
3380                 } else {
3381                         need_to_commit = B_TRUE;
3382                 }
3383         }
3384         if (res2 == Z_OK) {
3385                 if ((err = zonecfg_rm_aliased_rctl(handle, ALIAS_MAXSWAP))
3386                     != Z_OK) {
3387                         z_cmd_rt_perror(CMD_REMOVE, RT_MCAP, err, B_TRUE);
3388                         revert = B_TRUE;
3389                 } else {
3390                         need_to_commit = B_TRUE;
3391                 }
3392         }
3393         if (res3 == Z_OK) {
3394                 if ((err = zonecfg_rm_aliased_rctl(handle, ALIAS_MAXLOCKEDMEM))
3395                     != Z_OK) {
3396                         z_cmd_rt_perror(CMD_REMOVE, RT_MCAP, err, B_TRUE);
3397                         revert = B_TRUE;
3398                 } else {
3399                         need_to_commit = B_TRUE;
3400                 }
3401         }

3403         if (revert)
3404                 need_to_commit = B_FALSE;
3405 }
```

```
3407 static void
3408 remove_admin(cmd_t *cmd)
3409 {
3410         int err;

3412         /* traditional, qualified attr removal */
3413         if (cmd->cmd_prop_nv_pairs > 0) {
3414                 struct zone_admintab admintab;

3416                 if ((err = fill_in_admintab(cmd, &admintab, B_FALSE)) != Z_OK) {
3417                         z_cmd_rt_perror(CMD_REMOVE, RT_ADMIN,
3418                             err, B_TRUE);
3419                         return;
3420                 }
3421                 if ((err = zonecfg_delete_admin(handle, &admintab,
3422                     zone))
3423                     != Z_OK)
3424                         z_cmd_rt_perror(CMD_REMOVE, RT_ADMIN,
3425                             err, B_TRUE);
3426                 else
3427                         need_to_commit = B_TRUE;
3428                 return;
3429         } else {
3430                 /*
3431                  * unqualified admin removal.
3432                  * remove all admins but prompt if more
3433                  * than one.
3434                  */
3435                 if (!prompt_remove_resource(cmd, "admin"))
3436                         return;

3438                 if ((err = zonecfg_delete_admins(handle, zone))
3439                     != Z_OK)
3440                         z_cmd_rt_perror(CMD_REMOVE, RT_ADMIN,
3441                             err, B_TRUE);
3442                 else
3443                         need_to_commit = B_TRUE;
3444         }
3445 }

3447 static void
3448 remove_secflags()
3449 {
3450         int err;
3451         struct zone_secflagstab sectab = { 0 };

3453         if (zonecfg_lookup_secflags(handle, &sectab) != Z_OK) {
3454                 zerr("%s %s: %s", cmd_to_str(CMD_REMOVE),
3455                     rt_to_str(RT_SECFLAGS),
3456                     zonecfg_strerror(Z_NO_RESOURCE_TYPE));
3457                 return;
3458         }

3460         if ((err = zonecfg_delete_secflags(handle, &sectab)) != Z_OK) {
3461                 z_cmd_rt_perror(CMD_REMOVE, RT_SECFLAGS, err, B_TRUE);
3462                 return;
3463         }

3465         need_to_commit = B_TRUE;
3466 }

3468 static void
3469 remove_resource(cmd_t *cmd)
3470 {
3471         int type;
```



```
3472         int arg;
3473         boolean_t arg_err = B_FALSE;

3475         if ((type = cmd->cmd_res_type) == RT_UNKNOWN) {
3476                 long_usage(CMD_REMOVE, B_TRUE);
3477                 return;
3478         }

3480         optind = 0;
3481         while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "?F")) != EOF) {
3482                 switch (arg) {
3483                 case '?':
3484                         longer_usage(CMD_REMOVE);
3485                         arg_err = B_TRUE;
3486                         break;
3487                 case 'F':
3488                         break;
3489                 default:
3490                         short_usage(CMD_REMOVE);
3491                         arg_err = B_TRUE;
3492                         break;
3493                 }
3494         }
3495         if (arg_err)
3496                 return;

3498         if (initialize(B_TRUE) != Z_OK)
3499                 return;

3501         switch (type) {
3502         case RT_FS:
3503                 remove_fs(cmd);
3504                 return;
3505         case RT_NET:
3506                 remove_net(cmd);
3507                 return;
3508         case RT_DEVICE:
3509                 remove_device(cmd);
3510                 return;
3511         case RT_RCTL:
3512                 remove_rctl(cmd);
3513                 return;
3514         case RT_ATTR:
3515                 remove_attr(cmd);
3516                 return;
3517         case RT_DATASET:
3518                 remove_dataset(cmd);
3519                 return;
3520         case RT_DCPU:
3521                 remove_pset();
3522                 return;
3523         case RT_PCAP:
3524                 remove_pcap();
3525                 return;
3526         case RT_MCAP:
3527                 remove_mcap();
3528                 return;
3529         case RT_ADMIN:
3530                 remove_admin(cmd);
3531                 return;
3532         case RT_SECFLAGS:
3533                 remove_secflags();
3534                 return;
3535         default:
3536                 zone_perror(rt_to_str(type), Z_NO_RESOURCE_TYPE, B_TRUE);
3537                 long_usage(CMD_REMOVE, B_TRUE);
```

```
3538                 usage(B_FALSE, HELP_RESOURCES);
3539                 return;
3540         }
3541 }

3543 static void
3544 remove_property(cmd_t *cmd)
3545 {
3546         char *prop_id;
3547         int err, res_type, prop_type;
3548         property_value_ptr_t pp;
3549         struct zone_rctlvaltab *rctlvaltab;
3550         complex_property_ptr_t cx;

3552         res_type = resource_scope;
3553         prop_type = cmd->cmd_prop_name[0];
3554         if (res_type == RT_UNKNOWN || prop_type == PT_UNKNOWN) {
3555                 long_usage(CMD_REMOVE, B_TRUE);
3556                 return;
3557         }

3559         if (cmd->cmd_prop_nv_pairs != 1) {
3560                 long_usage(CMD_ADD, B_TRUE);
3561                 return;
3562         }

3564         if (initialize(B_TRUE) != Z_OK)
3565                 return;

3567         switch (res_type) {
3568         case RT_FS:
3569                 if (prop_type != PT_OPTIONS) {
3570                         zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
3571                             B_TRUE);
3572                         long_usage(CMD_REMOVE, B_TRUE);
3573                         usage(B_FALSE, HELP_PROPS);
3574                         return;
3575                 }
3576                 pp = cmd->cmd_property_ptr[0];
3577                 if (pp->pv_type == PROP_VAL_COMPLEX) {
3578                         zerr(gettext("A %s or %s value was expected here."),
3579                             pvt_to_str(PROP_VAL_SIMPLE),
3580                             pvt_to_str(PROP_VAL_LIST));
3581                         saw_error = B_TRUE;
3582                         return;
3583                 }
3584                 if (pp->pv_type == PROP_VAL_SIMPLE) {
3585                         if (pp->pv_simple == NULL) {
3586                                 long_usage(CMD_ADD, B_TRUE);
3587                                 return;
3588                         }
3589                         prop_id = pp->pv_simple;
3590                         err = zonecfg_remove_fs_option(&in_progress_fstab,
3591                             prop_id);
3592                         if (err != Z_OK)
3593                                 zone_perror(pt_to_str(prop_type), err, B_TRUE);
3594                 } else {
3595                         list_property_ptr_t list;

3597                         for (list = pp->pv_list; list != NULL;
3598                             list = list->lp_next) {
3599                                 prop_id = list->lp_simple;
3600                                 if (prop_id == NULL)
3601                                         break;
3602                                 err = zonecfg_remove_fs_option(
3603                                     &in_progress_fstab, prop_id);
```



```
3604                                 if (err != Z_OK)
3605                                         zone_perror(pt_to_str(prop_type), err,
3606                                             B_TRUE);
3607                         }
3608                 }
3609                 return;
3610         case RT_RCTL:
3611                 if (prop_type != PT_VALUE) {
3612                         zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
3613                             B_TRUE);
3614                         long_usage(CMD_REMOVE, B_TRUE);
3615                         usage(B_FALSE, HELP_PROPS);
3616                         return;
3617                 }
3618                 pp = cmd->cmd_property_ptr[0];
3619                 if (pp->pv_type != PROP_VAL_COMPLEX) {
3620                         zerr(gettext("A %s value was expected here."),
3621                             pvt_to_str(PROP_VAL_COMPLEX));
3622                         saw_error = B_TRUE;
3623                         return;
3624                 }
3625                 if ((rctlvaltab = alloc_rctlvaltab()) == NULL) {
3626                         zone_perror(zone, Z_NOMEM, B_TRUE);
3627                         exit(Z_ERR);
3628                 }
3629                 for (cx = pp->pv_complex; cx != NULL; cx = cx->cp_next) {
3630                         switch (cx->cp_type) {
3631                         case PT_PRIV:
3632                                 (void) strlcpy(rctlvaltab->zone_rctlval_priv,
3633                                     cx->cp_value,
3634                                     sizeof (rctlvaltab->zone_rctlval_priv));
3635                                 break;
3636                         case PT_LIMIT:
3637                                 (void) strlcpy(rctlvaltab->zone_rctlval_limit,
3638                                     cx->cp_value,
3639                                     sizeof (rctlvaltab->zone_rctlval_limit));
3640                                 break;
3641                         case PT_ACTION:
3642                                 (void) strlcpy(rctlvaltab->zone_rctlval_action,
3643                                     cx->cp_value,
3644                                     sizeof (rctlvaltab->zone_rctlval_action));
3645                                 break;
3646                         default:
3647                                 zone_perror(pt_to_str(prop_type),
3648                                     Z_NO_PROPERTY_TYPE, B_TRUE);
3649                                 long_usage(CMD_ADD, B_TRUE);
3650                                 usage(B_FALSE, HELP_PROPS);
3651                                 zonecfg_free_rctl_value_list(rctlvaltab);
3652                                 return;
3653                         }
3654                 }
3655                 rctlvaltab->zone_rctlval_next = NULL;
3656                 err = zonecfg_remove_rctl_value(&in_progress_rctltab,
3657                     rctlvaltab);
3658                 if (err != Z_OK)
3659                         zone_perror(pt_to_str(prop_type), err, B_TRUE);
3660                 zonecfg_free_rctl_value_list(rctlvaltab);
3661                 return;
3662         case RT_NET:
3663                 if (prop_type != PT_DEFROUTER) {
3664                         zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
3665                             B_TRUE);
3666                         long_usage(CMD_REMOVE, B_TRUE);
3667                         usage(B_FALSE, HELP_PROPS);
3668                         return;
3669                 } else {
```

```
3670                        bzero(&in_progress_nwiftab.zone_nwif_defrouter,
3671                            sizeof (in_progress_nwiftab.zone_nwif_defrouter));
3672                        return;
3673                }
3674        default:
3675                zone_perror(rt_to_str(res_type), Z_NO_RESOURCE_TYPE, B_TRUE);
3676                long_usage(CMD_REMOVE, B_TRUE);
3677                usage(B_FALSE, HELP_RESOURCES);
3678                return;
3679        }
3680 }

3682 void
3683 remove_func(cmd_t *cmd)
3684 {
3685        if (zone_is_read_only(CMD_REMOVE))
3686                return;

3688        assert(cmd != NULL);

3690        if (global_scope) {
3691                if (gz_invalid_resource(cmd->cmd_res_type)) {
3692                        zerr(gettext("%s is not a valid resource for the "
3693                            "global zone."), rt_to_str(cmd->cmd_res_type));
3694                        saw_error = B_TRUE;
3695                        return;
3696                }
3697                remove_resource(cmd);
3698        } else {
3699                remove_property(cmd);
3700        }
3701 }

3703 static void
3704 clear_property(cmd_t *cmd)
3705 {
3706        int res_type, prop_type;

3708        res_type = resource_scope;
3709        prop_type = cmd->cmd_res_type;
3710        if (res_type == RT_UNKNOWN || prop_type == PT_UNKNOWN) {
3711                long_usage(CMD_CLEAR, B_TRUE);
3712                return;
3713        }

3715        if (initialize(B_TRUE) != Z_OK)
3716                return;

3718        switch (res_type) {
3719        case RT_FS:
3720                if (prop_type == PT_RAW) {
3721                        in_progress_fstab.zone_fs_raw[0] = '\0';
3722                        need_to_commit = B_TRUE;
3723                        return;
3724                }
3725                break;
3726        case RT_DCPU:
3727                if (prop_type == PT_IMPORTANCE) {
3728                        in_progress_psettab.zone_importance[0] = '\0';
3729                        need_to_commit = B_TRUE;
3730                        return;
3731                }
3732                break;
3733        case RT_MCAP:
3734                switch (prop_type) {
3735                case PT_PHYSICAL:
```



```
3736                        in_progress_mcaptab.zone_physmem_cap[0] = '\0';
3737                        need_to_commit = B_TRUE;
3738                        return;
3739                case PT_SWAP:
3740                        remove_aliased_rctl(PT_SWAP, ALIAS_MAXSWAP);
3741                        return;
3742                case PT_LOCKED:
3743                        remove_aliased_rctl(PT_LOCKED, ALIAS_MAXLOCKEDMEM);
3744                        return;
3745                }
3746                break;
3747        case RT_SECFLAGS:
3748                switch (prop_type) {
3749                case PT_LOWER:
3750                        in_progress_secflagstab.zone_secflags_lower[0] = '\0';
3751                        need_to_commit = B_TRUE;
3752                        return;
3753                case PT_DEFAULT:
3754                        in_progress_secflagstab.zone_secflags_default[0] = '\0';
3755                        need_to_commit = B_TRUE;
3756                        return;
3757                case PT_UPPER:
3758                        in_progress_secflagstab.zone_secflags_upper[0] = '\0';
3759                        need_to_commit = B_TRUE;
3760                        return;
3761                }
3762                break;
3763        default:
3764                break;
3765        }

3767        zone_perror(pt_to_str(prop_type), Z_CLEAR_DISALLOW, B_TRUE);
3768 }

3770 static void
3771 clear_global(cmd_t *cmd)
3772 {
3773        int err, type;

3775        if ((type = cmd->cmd_res_type) == RT_UNKNOWN) {
3776                long_usage(CMD_CLEAR, B_TRUE);
3777                return;
3778        }

3780        if (initialize(B_TRUE) != Z_OK)
3781                return;

3783        switch (type) {
3784        case PT_ZONENAME:
3785                /* FALLTHRU */
3786        case PT_ZONEPATH:
3787                /* FALLTHRU */
3788        case PT_BRAND:
3789                zone_perror(pt_to_str(type), Z_CLEAR_DISALLOW, B_TRUE);
3790                return;
3791        case PT_AUTOBOOT:
3792                /* false is default; we'll treat as equivalent to clearing */
3793                if ((err = zonecfg_set_autoboot(handle, B_FALSE)) != Z_OK)
3794                        z_cmd_rt_perror(CMD_CLEAR, RT_AUTOBOOT, err, B_TRUE);
3795                else
3796                        need_to_commit = B_TRUE;
3797                return;
3798        case PT_POOL:
3799                if ((err = zonecfg_set_pool(handle, NULL)) != Z_OK)
3800                        z_cmd_rt_perror(CMD_CLEAR, RT_POOL, err, B_TRUE);
3801                else
```

```
3802                        need_to_commit = B_TRUE;
3803                return;
3804        case PT_LIMITPRIV:
3805                if ((err = zonecfg_set_limitpriv(handle, NULL)) != Z_OK)
3806                        z_cmd_rt_perror(CMD_CLEAR, RT_LIMITPRIV, err, B_TRUE);
3807                else
3808                        need_to_commit = B_TRUE;
3809                return;
3810        case PT_BOOTARGS:
3811                if ((err = zonecfg_set_bootargs(handle, NULL)) != Z_OK)
3812                        z_cmd_rt_perror(CMD_CLEAR, RT_BOOTARGS, err, B_TRUE);
3813                else
3814                        need_to_commit = B_TRUE;
3815                return;
3816        case PT_SCHED:
3817                if ((err = zonecfg_set_sched(handle, NULL)) != Z_OK)
3818                        z_cmd_rt_perror(CMD_CLEAR, RT_SCHED, err, B_TRUE);
3819                else
3820                        need_to_commit = B_TRUE;
3821                return;
3822        case PT_IPTYPE:
3823                /* shared is default; we'll treat as equivalent to clearing */
3824                if ((err = zonecfg_set_iptype(handle, ZS_SHARED)) != Z_OK)
3825                        z_cmd_rt_perror(CMD_CLEAR, RT_IPTYPE, err, B_TRUE);
3826                else
3827                        need_to_commit = B_TRUE;
3828                return;
3829        case PT_MAXLWPS:
3830                remove_aliased_rctl(PT_MAXLWPS, ALIAS_MAXLWPS);
3831                return;
3832        case PT_MAXPROCS:
3833                remove_aliased_rctl(PT_MAXPROCS, ALIAS_MAXPROCS);
3834                return;
3835        case PT_MAXSHMMEM:
3836                remove_aliased_rctl(PT_MAXSHMMEM, ALIAS_MAXSHMMEM);
3837                return;
3838        case PT_MAXSHMIDS:
3839                remove_aliased_rctl(PT_MAXSHMIDS, ALIAS_MAXSHMIDS);
3840                return;
3841        case PT_MAXMSGIDS:
3842                remove_aliased_rctl(PT_MAXMSGIDS, ALIAS_MAXMSGIDS);
3843                return;
3844        case PT_MAXSEMIDS:
3845                remove_aliased_rctl(PT_MAXSEMIDS, ALIAS_MAXSEMIDS);
3846                return;
3847        case PT_SHARES:
3848                remove_aliased_rctl(PT_SHARES, ALIAS_SHARES);
3849                return;
3850        case PT_HOSTID:
3851                if ((err = zonecfg_set_hostid(handle, NULL)) != Z_OK)
3852                        z_cmd_rt_perror(CMD_CLEAR, RT_HOSTID, err, B_TRUE);
3853                else
3854                        need_to_commit = B_TRUE;
3855                return;
3856        case PT_FS_ALLOWED:
3857                if ((err = zonecfg_set_fs_allowed(handle, NULL)) != Z_OK)
3858                        z_cmd_rt_perror(CMD_CLEAR, RT_FS_ALLOWED, err, B_TRUE);
3859                else
3860                        need_to_commit = B_TRUE;
3861                return;
3862        default:
3863                zone_perror(pt_to_str(type), Z_NO_PROPERTY_TYPE, B_TRUE);
3864                long_usage(CMD_CLEAR, B_TRUE);
3865                usage(B_FALSE, HELP_PROPS);
3866                return;
3867        }
```



```
3868 }

3870 void
3871 clear_func(cmd_t *cmd)
3872 {
3873         if (zone_is_read_only(CMD_CLEAR))
3874                 return;

3876         assert(cmd != NULL);

3878         if (global_scope) {
3879                 if (gz_invalid_property(cmd->cmd_res_type)) {
3880                         zerr(gettext("%s is not a valid property for the "
3881                             "global zone."), pt_to_str(cmd->cmd_res_type));
3882                         saw_error = B_TRUE;
3883                         return;
3884                 }

3886                 clear_global(cmd);
3887         } else {
3888                 clear_property(cmd);
3889         }
3890 }

3892 void
3893 select_func(cmd_t *cmd)
3894 {
3895         int type, err, res;
3896         uint64_t limit;
3897         uint64_t tmp;

3899         if (zone_is_read_only(CMD_SELECT))
3900                 return;

3902         assert(cmd != NULL);

3904         if (global_scope) {
3905                 global_scope = B_FALSE;
3906                 resource_scope = cmd->cmd_res_type;
3907                 end_op = CMD_SELECT;
3908         } else {
3909                 scope_usage(CMD_SELECT);
3910                 return;
3911         }

3913         if ((type = cmd->cmd_res_type) == RT_UNKNOWN) {
3914                 long_usage(CMD_SELECT, B_TRUE);
3915                 return;
3916         }

3918         if (initialize(B_TRUE) != Z_OK)
3919                 return;

3921         switch (type) {
3922         case RT_FS:
3923                 if ((err = fill_in_fstab(cmd, &old_fstab, B_FALSE)) != Z_OK) {
3924                         z_cmd_rt_perror(CMD_SELECT, RT_FS, err, B_TRUE);
3925                         global_scope = B_TRUE;
3926                 }
3927                 bcopy(&old_fstab, &in_progress_fstab,
3928                     sizeof (struct zone_fstab));
3929                 return;
3930         case RT_NET:
3931                 if ((err = fill_in_nwiftab(cmd, &old_nwiftab, B_FALSE))
3932                     != Z_OK) {
3933                         z_cmd_rt_perror(CMD_SELECT, RT_NET, err, B_TRUE);
```

```
3934                        global_scope = B_TRUE;
3935                }
3936                bcopy(&old_nwiftab, &in_progress_nwiftab,
3937                    sizeof (struct zone_nwiftab));
3938                return;
3939        case RT_DEVICE:
3940                if ((err = fill_in_devtab(cmd, &old_devtab, B_FALSE)) != Z_OK) {
3941                        z_cmd_rt_perror(CMD_SELECT, RT_DEVICE, err, B_TRUE);
3942                        global_scope = B_TRUE;
3943                }
3944                bcopy(&old_devtab, &in_progress_devtab,
3945                    sizeof (struct zone_devtab));
3946                return;
3947        case RT_RCTL:
3948                if ((err = fill_in_rctltab(cmd, &old_rctltab, B_FALSE))
3949                    != Z_OK) {
3950                        z_cmd_rt_perror(CMD_SELECT, RT_RCTL, err, B_TRUE);
3951                        global_scope = B_TRUE;
3952                }
3953                bcopy(&old_rctltab, &in_progress_rctltab,
3954                    sizeof (struct zone_rctltab));
3955                return;
3956        case RT_ATTR:
3957                if ((err = fill_in_attrtab(cmd, &old_attrtab, B_FALSE))
3958                    != Z_OK) {
3959                        z_cmd_rt_perror(CMD_SELECT, RT_ATTR, err, B_TRUE);
3960                        global_scope = B_TRUE;
3961                }
3962                bcopy(&old_attrtab, &in_progress_attrtab,
3963                    sizeof (struct zone_attrtab));
3964                return;
3965        case RT_DATASET:
3966                if ((err = fill_in_dstab(cmd, &old_dstab, B_FALSE)) != Z_OK) {
3967                        z_cmd_rt_perror(CMD_SELECT, RT_DATASET, err, B_TRUE);
3968                        global_scope = B_TRUE;
3969                }
3970                bcopy(&old_dstab, &in_progress_dstab,
3971                    sizeof (struct zone_dstab));
3972                return;
3973        case RT_DCPU:
3974                if ((err = zonecfg_lookup_pset(handle, &old_psettab)) != Z_OK) {
3975                        z_cmd_rt_perror(CMD_SELECT, RT_DCPU, err, B_TRUE);
3976                        global_scope = B_TRUE;
3977                }
3978                bcopy(&old_psettab, &in_progress_psettab,
3979                    sizeof (struct zone_psettab));
3980                return;
3981        case RT_PCAP:
3982                if ((err = zonecfg_get_aliased_rctl(handle, ALIAS_CPUCAP, &tmp))
3983                    != Z_OK) {
3984                        z_cmd_rt_perror(CMD_SELECT, RT_PCAP, err, B_TRUE);
3985                        global_scope = B_TRUE;
3986                }
3987                return;
3988        case RT_MCAP:
3989                /* if none of these exist, there is no resource to select */
3990                if ((res = zonecfg_lookup_mcap(handle, &old_mcaptab)) != Z_OK &&
3991                    zonecfg_get_aliased_rctl(handle, ALIAS_MAXSWAP, &limit)
3992                    != Z_OK &&
3993                    zonecfg_get_aliased_rctl(handle, ALIAS_MAXLOCKEDMEM, &limit)
3994                    != Z_OK) {
3995                        z_cmd_rt_perror(CMD_SELECT, RT_MCAP, Z_NO_RESOURCE_TYPE,
3996                            B_TRUE);
3997                        global_scope = B_TRUE;
3998                }
3999                if (res == Z_OK)
```



```
4000                        bcopy(&old_mcaptab, &in_progress_mcaptab,
4001                            sizeof (struct zone_mcaptab));
4002                else
4003                        bzero(&in_progress_mcaptab,
4004                            sizeof (in_progress_mcaptab));
4005                return;
4006        case RT_ADMIN:
4007                if ((err = fill_in_admintab(cmd, &old_admintab, B_FALSE))
4008                    != Z_OK) {
4009                        z_cmd_rt_perror(CMD_SELECT, RT_ADMIN, err,
4010                            B_TRUE);
4011                        global_scope = B_TRUE;
4012                }
4013                bcopy(&old_admintab, &in_progress_admintab,
4014                    sizeof (struct zone_admintab));
4015                return;
4016        case RT_SECFLAGS:
4017                if ((err = fill_in_secflagstab(cmd, &old_secflagstab, B_FALSE))
4018                    != Z_OK) {
4019                        z_cmd_rt_perror(CMD_SELECT, RT_SECFLAGS, err,
4020                            B_TRUE);
4021                        global_scope = B_TRUE;
4022                }
4023                bcopy(&old_secflagstab, &in_progress_secflagstab,
4024                    sizeof (struct zone_secflagstab));
4025                return;
4026        default:
4027                zone_perror(rt_to_str(type), Z_NO_RESOURCE_TYPE, B_TRUE);
4028                long_usage(CMD_SELECT, B_TRUE);
4029                usage(B_FALSE, HELP_RESOURCES);
4030                return;
4031        }
4032 }

4034 /*
4035  * Network "addresses" can be one of the following forms:
4036  *      <IPv4 address>
4037  *      <IPv4 address>/<prefix length>
4038  *      <IPv6 address>/<prefix length>
4039  *      <host name>
4040  *      <host name>/<prefix length>
4041  * In other words, the "/" followed by a prefix length is allowed but not
4042  * required for IPv4 addresses and host names, and required for IPv6 addresses.
4043  * If a prefix length is given, it must be in the allowable range: 0 to 32 for
4044  * IPv4 addresses and host names, 0 to 128 for IPv6 addresses.
4045  * Host names must start with an alpha-numeric character, and all subsequent
4046  * characters must be either alpha-numeric or "-".
4047  *
4048  * In some cases, e.g., the nexthop for the defrouter, the context indicates
4049  * that this is the IPV4_ABITS or IPV6_ABITS netmask, in which case we don't
4050  * require the /<prefix length> (and should ignore it if provided).
4051  */

4053 static int
4054 validate_net_address_syntax(char *address, boolean_t ishost)
4055 {
4056        char *slashp, part1[MAXHOSTNAMELEN];
4057        struct in6_addr in6;
4058        struct in_addr in4;
4059        int prefixlen, i;

4061        /*
4062         * Copy the part before any '/' into part1 or copy the whole
4063         * thing if there is no '/'.
4064         */
4065        if ((slashp = strchr(address, '/')) != NULL) {
```

```
4066                *slashp = '\0';
4067                (void) strlcpy(part1, address, sizeof (part1));
4068                *slashp = '/';
4069                prefixlen = atoi(++slashp);
4070        } else {
4071                (void) strlcpy(part1, address, sizeof (part1));
4072        }

4074        if (ishost && slashp != NULL) {
4075                zerr(gettext("Warning: prefix length in %s is not required and "
4076                    "will be ignored. The default host-prefix length "
4077                    "will be used"), address);
4078        }


4081        if (inet_pton(AF_INET6, part1, &in6) == 1) {
4082                if (ishost) {
4083                        prefixlen = IPV6_ABITS;
4084                } else if (slashp == NULL) {
4085                        zerr(gettext("%s: IPv6 addresses "
4086                            "require /prefix-length suffix."), address);
4087                        return (Z_ERR);
4088                }
4089                if (prefixlen < 0 || prefixlen > 128) {
4090                        zerr(gettext("%s: IPv6 address "
4091                            "prefix lengths must be 0 - 128."), address);
4092                        return (Z_ERR);
4093                }
4094                return (Z_OK);
4095        }

4097        /* At this point, any /prefix must be for IPv4. */
4098        if (ishost)
4099                prefixlen = IPV4_ABITS;
4100        else if (slashp != NULL) {
4101                if (prefixlen < 0 || prefixlen > 32) {
4102                        zerr(gettext("%s: IPv4 address "
4103                            "prefix lengths must be 0 - 32."), address);
4104                        return (Z_ERR);
4105                }
4106        }

4108        if (inet_pton(AF_INET, part1, &in4) == 1)
4109                return (Z_OK);

4111        /* address may also be a host name */
4112        if (!isalnum(part1[0])) {
4113                zerr(gettext("%s: bogus host name or network address syntax"),
4114                    part1);
4115                saw_error = B_TRUE;
4116                usage(B_FALSE, HELP_NETADDR);
4117                return (Z_ERR);
4118        }
4119        for (i = 1; part1[i]; i++)
4120                if (!isalnum(part1[i]) && part1[i] != '-' && part1[i] != '.') {
4121                        zerr(gettext("%s: bogus host name or "
4122                            "network address syntax"), part1);
4123                        saw_error = B_TRUE;
4124                        usage(B_FALSE, HELP_NETADDR);
4125                        return (Z_ERR);
4126                }
4127        return (Z_OK);
4128 }

4130 static int
4131 validate_net_physical_syntax(const char *ifname)
```



```
4132 {
4133        ifspec_t ifnameprop;
4134        zone_iptype_t iptype;

4136        if (zonecfg_get_iptype(handle, &iptype) != Z_OK) {
4137                zerr(gettext("zone configuration has an invalid or nonexistent "
4138                    "ip-type property"));
4139                return (Z_ERR);
4140        }
4141        switch (iptype) {
4142        case ZS_SHARED:
4143                if (ifparse_ifspec(ifname, &ifnameprop) == B_FALSE) {
4144                        zerr(gettext("%s: invalid physical interface name"),
4145                            ifname);
4146                        return (Z_ERR);
4147                }
4148                if (ifnameprop.ifsp_lunvalid) {
4149                        zerr(gettext("%s: LUNs not allowed in physical "
4150                            "interface names"), ifname);
4151                        return (Z_ERR);
4152                }
4153                break;
4154        case ZS_EXCLUSIVE:
4155                if (dladm_valid_linkname(ifname) == B_FALSE) {
4156                        if (strchr(ifname, ':') != NULL)
4157                                zerr(gettext("%s: physical interface name "
4158                                    "required; logical interface name not "
4159                                    "allowed"), ifname);
4160                        else
4161                                zerr(gettext("%s: invalid physical interface "
4162                                    "name"), ifname);
4163                        return (Z_ERR);
4164                }
4165                break;
4166        }
4167        return (Z_OK);
4168 }

4170 static boolean_t
4171 valid_fs_type(const char *type)
4172 {
4173        /*
4174         * Is this a valid path component?
4175         */
4176        if (strlen(type) + 1 > MAXNAMELEN)
4177                return (B_FALSE);
4178        /*
4179         * Make sure a bad value for "type" doesn't make
4180         * /usr/lib/fs/<type>/mount turn into something else.
4181         */
4182        if (strchr(type, '/') != NULL || type[0] == '\0' ||
4183            strcmp(type, ".") == 0 || strcmp(type, "..") == 0)
4184                return (B_FALSE);
4185        /*
4186         * More detailed verification happens later by zoneadm(1m).
4187         */
4188        return (B_TRUE);
4189 }

4191 static boolean_t
4192 allow_exclusive()
4193 {
4194        brand_handle_t  bh;
4195        char            brand[MAXNAMELEN];
4196        boolean_t       ret;
```

```
4198         if (zonecfg_get_brand(handle, brand, sizeof (brand)) != Z_OK) {
4199                 zerr("%s: %s\n", zone, gettext("could not get zone brand"));
4200                 return (B_FALSE);
4201         }
4202         if ((bh = brand_open(brand)) == NULL) {
4203                 zerr("%s: %s\n", zone, gettext("unknown brand."));
4204                 return (B_FALSE);
4205         }
4206         ret = brand_allow_exclusive_ip(bh);
4207         brand_close(bh);
4208         if (!ret)
4209                 zerr(gettext("%s cannot be '%s' when %s is '%s'."),
4210                     pt_to_str(PT_IPTYPE), "exclusive",
4211                     pt_to_str(PT_BRAND), brand);
4212         return (ret);
4213 }

4215 static void
4216 set_aliased_rctl(char *alias, int prop_type, char *s)
4217 {
4218         uint64_t limit;
4219         int err;
4220         char tmp[128];

4222         if (global_zone && strcmp(alias, ALIAS_SHARES) != 0)
4223                 zerr(gettext("WARNING: Setting a global zone resource "
4224                     "control too low could deny\nservice "
4225                     "to even the root user; "
4226                     "this could render the system impossible\n"
4227                     "to administer.  Please use caution."));

4229         /* convert memory based properties */
4230         if (prop_type == PT_MAXSHMMEM) {
4231                 if (!zonecfg_valid_memlimit(s, &limit)) {
4232                         zerr(gettext("A non-negative number with a required "
4233                             "scale suffix (K, M, G or T) was expected\nhere."));
4234                         saw_error = B_TRUE;
4235                         return;
4236                 }

4238                 (void) snprintf(tmp, sizeof (tmp), "%llu", limit);
4239                 s = tmp;
4240         }

4242         if (!zonecfg_aliased_rctl_ok(handle, alias)) {
4243                 zone_perror(pt_to_str(prop_type), Z_ALIAS_DISALLOW, B_FALSE);
4244                 saw_error = B_TRUE;
4245         } else if (!zonecfg_valid_alias_limit(alias, s, &limit)) {
4246                 zerr(gettext("%s property is out of range."),
4247                     pt_to_str(prop_type));
4248                 saw_error = B_TRUE;
4249         } else if ((err = zonecfg_set_aliased_rctl(handle, alias, limit))
4250             != Z_OK) {
4251                 zone_perror(zone, err, B_TRUE);
4252                 saw_error = B_TRUE;
4253         } else {
4254                 need_to_commit = B_TRUE;
4255         }
4256 }

4258 static void
4259 set_in_progress_nwiftab_address(char *prop_id, int prop_type)
4260 {
4261         if (prop_type == PT_ADDRESS) {
4262                 (void) strlcpy(in_progress_nwiftab.zone_nwif_address, prop_id,
4263                     sizeof (in_progress_nwiftab.zone_nwif_address));
```



```
4264         } else {
4265                 assert(prop_type == PT_ALLOWED_ADDRESS);
4266                 (void) strlcpy(in_progress_nwiftab.zone_nwif_allowed_address,
4267                     prop_id,
4268                     sizeof (in_progress_nwiftab.zone_nwif_allowed_address));
4269         }
4270 }

4272 void
4273 set_func(cmd_t *cmd)
4274 {
4275         char *prop_id;
4276         int arg, err, res_type, prop_type;
4277         property_value_ptr_t pp;
4278         boolean_t autoboot;
4279         zone_iptype_t iptype;
4280         boolean_t force_set = B_FALSE;
4281         size_t physmem_size = sizeof (in_progress_mcaptab.zone_physmem_cap);
4282         uint64_t mem_cap, mem_limit;
4283         float cap;
4284         char *unitp;
4285         struct zone_psettab tmp_psettab;
4286         boolean_t arg_err = B_FALSE;

4288         if (zone_is_read_only(CMD_SET))
4289                 return;

4291         assert(cmd != NULL);

4293         optind = opterr = 0;
4294         while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "F")) != EOF) {
4295                 switch (arg) {
4296                 case 'F':
4297                         force_set = B_TRUE;
4298                         break;
4299                 default:
4300                         if (optopt == '?')
4301                                 longer_usage(CMD_SET);
4302                         else
4303                                 short_usage(CMD_SET);
4304                         arg_err = B_TRUE;
4305                         break;
4306                 }
4307         }
4308         if (arg_err)
4309                 return;

4311         prop_type = cmd->cmd_prop_name[0];
4312         if (global_scope) {
4313                 if (gz_invalid_property(prop_type)) {
4314                         zerr(gettext("%s is not a valid property for the "
4315                             "global zone."), pt_to_str(prop_type));
4316                         saw_error = B_TRUE;
4317                         return;
4318                 }

4320                 if (prop_type == PT_ZONENAME) {
4321                         res_type = RT_ZONENAME;
4322                 } else if (prop_type == PT_ZONEPATH) {
4323                         res_type = RT_ZONEPATH;
4324                 } else if (prop_type == PT_AUTOBOOT) {
4325                         res_type = RT_AUTOBOOT;
4326                 } else if (prop_type == PT_BRAND) {
4327                         res_type = RT_BRAND;
4328                 } else if (prop_type == PT_POOL) {
4329                         res_type = RT_POOL;
```

```
4330                } else if (prop_type == PT_LIMITPRIV) {
4331                        res_type = RT_LIMITPRIV;
4332                } else if (prop_type == PT_BOOTARGS) {
4333                        res_type = RT_BOOTARGS;
4334                } else if (prop_type == PT_SCHED) {
4335                        res_type = RT_SCHED;
4336                } else if (prop_type == PT_IPTYPE) {
4337                        res_type = RT_IPTYPE;
4338                } else if (prop_type == PT_MAXLWPS) {
4339                        res_type = RT_MAXLWPS;
4340                } else if (prop_type == PT_MAXPROCS) {
4341                        res_type = RT_MAXPROCS;
4342                } else if (prop_type == PT_MAXSHMMEM) {
4343                        res_type = RT_MAXSHMMEM;
4344                } else if (prop_type == PT_MAXSHMIDS) {
4345                        res_type = RT_MAXSHMIDS;
4346                } else if (prop_type == PT_MAXMSGIDS) {
4347                        res_type = RT_MAXMSGIDS;
4348                } else if (prop_type == PT_MAXSEMIDS) {
4349                        res_type = RT_MAXSEMIDS;
4350                } else if (prop_type == PT_SHARES) {
4351                        res_type = RT_SHARES;
4352                } else if (prop_type == PT_HOSTID) {
4353                        res_type = RT_HOSTID;
4354                } else if (prop_type == PT_FS_ALLOWED) {
4355                        res_type = RT_FS_ALLOWED;
4356                } else {
4357                        zerr(gettext("Cannot set a resource-specific property "
4358                            "from the global scope."));
4359                        saw_error = B_TRUE;
4360                        return;
4361                }
4362        } else {
4363                res_type = resource_scope;
4364        }

4366        if (force_set) {
4367                if (res_type != RT_ZONEPATH) {
4368                        zerr(gettext("Only zonepath setting can be forced."));
4369                        saw_error = B_TRUE;
4370                        return;
4371                }
4372                if (!zonecfg_in_alt_root()) {
4373                        zerr(gettext("Zonepath is changeable only in an "
4374                            "alternate root."));
4375                        saw_error = B_TRUE;
4376                        return;
4377                }
4378        }

4380        pp = cmd->cmd_property_ptr[0];
4381        /*
4382         * A nasty expression but not that complicated:
4383         * 1. fs options are simple or list (tested below)
4384         * 2. rctl value's are complex or list (tested below)
4385         * Anything else should be simple.
4386         */
4387        if (!(res_type == RT_FS && prop_type == PT_OPTIONS) &&
4388            !(res_type == RT_RCTL && prop_type == PT_VALUE) &&
4389            (pp->pv_type != PROP_VAL_SIMPLE ||
4390            (prop_id = pp->pv_simple) == NULL)) {
4391                zerr(gettext("A %s value was expected here."),
4392                    pvt_to_str(PROP_VAL_SIMPLE));
4393                saw_error = B_TRUE;
4394                return;
4395        }
```



```
4396        if (prop_type == PT_UNKNOWN) {
4397                long_usage(CMD_SET, B_TRUE);
4398                return;
4399        }

4401        /*
4402         * Special case: the user can change the zone name prior to 'create';
4403         * if the zone already exists, we fall through letting initialize()
4404         * and the rest of the logic run.
4405         */
4406        if (res_type == RT_ZONENAME && got_handle == B_FALSE &&
4407            !state_atleast(ZONE_STATE_CONFIGURED)) {
4408                if ((err = zonecfg_validate_zonename(prop_id)) != Z_OK) {
4409                        zone_perror(prop_id, err, B_TRUE);
4410                        usage(B_FALSE, HELP_SYNTAX);
4411                        return;
4412                }
4413                (void) strlcpy(zone, prop_id, sizeof (zone));
4414                return;
4415        }

4417        if (initialize(B_TRUE) != Z_OK)
4418                return;

4420        switch (res_type) {
4421        case RT_ZONENAME:
4422                if ((err = zonecfg_set_name(handle, prop_id)) != Z_OK) {
4423                        /*
4424                         * Use prop_id instead of 'zone' here, since we're
4425                         * reporting a problem about the *new* zonename.
4426                         */
4427                        zone_perror(prop_id, err, B_TRUE);
4428                        usage(B_FALSE, HELP_SYNTAX);
4429                } else {
4430                        need_to_commit = B_TRUE;
4431                        (void) strlcpy(zone, prop_id, sizeof (zone));
4432                }
4433                return;
4434        case RT_ZONEPATH:
4435                if (!force_set && state_atleast(ZONE_STATE_INSTALLED)) {
4436                        zerr(gettext("Zone %s already installed; %s %s not "
4437                            "allowed."), zone, cmd_to_str(CMD_SET),
4438                            rt_to_str(RT_ZONEPATH));
4439                        return;
4440                }
4441                if (validate_zonepath_syntax(prop_id) != Z_OK) {
4442                        saw_error = B_TRUE;
4443                        return;
4444                }
4445                if ((err = zonecfg_set_zonepath(handle, prop_id)) != Z_OK)
4446                        zone_perror(zone, err, B_TRUE);
4447                else
4448                        need_to_commit = B_TRUE;
4449                return;
4450        case RT_BRAND:
4451                if (state_atleast(ZONE_STATE_INSTALLED)) {
4452                        zerr(gettext("Zone %s already installed; %s %s not "
4453                            "allowed."), zone, cmd_to_str(CMD_SET),
4454                            rt_to_str(RT_BRAND));
4455                        return;
4456                }
4457                if ((err = zonecfg_set_brand(handle, prop_id)) != Z_OK)
4458                        zone_perror(zone, err, B_TRUE);
4459                else
4460                        need_to_commit = B_TRUE;
4461                return;
```

```
4462         case RT_AUTOBOOT:
4463                 if (strcmp(prop_id, "true") == 0) {
4464                         autoboot = B_TRUE;
4465                 } else if (strcmp(prop_id, "false") == 0) {
4466                         autoboot = B_FALSE;
4467                 } else {
4468                         zerr(gettext("%s value must be '%s' or '%s'."),
4469                             pt_to_str(PT_AUTOBOOT), "true", "false");
4470                         saw_error = B_TRUE;
4471                         return;
4472                 }
4473                 if ((err = zonecfg_set_autoboot(handle, autoboot)) != Z_OK)
4474                         zone_perror(zone, err, B_TRUE);
4475                 else
4476                         need_to_commit = B_TRUE;
4477                 return;
4478         case RT_POOL:
4479                 /* don't allow use of the reserved temporary pool names */
4480                 if (strncmp("SUNW", prop_id, 4) == 0) {
4481                         zerr(gettext("pool names starting with SUNW are "
4482                             "reserved."));
4483                         saw_error = B_TRUE;
4484                         return;
4485                 }

4487                 /* can't set pool if dedicated-cpu exists */
4488                 if (zonecfg_lookup_pset(handle, &tmp_psettab) == Z_OK) {
4489                         zerr(gettext("The %s resource already exists.  "
4490                             "A persistent pool is incompatible\nwith the %s "
4491                             "resource."), rt_to_str(RT_DCPU),
4492                             rt_to_str(RT_DCPU));
4493                         saw_error = B_TRUE;
4494                         return;
4495                 }

4497                 if ((err = zonecfg_set_pool(handle, prop_id)) != Z_OK)
4498                         zone_perror(zone, err, B_TRUE);
4499                 else
4500                         need_to_commit = B_TRUE;
4501                 return;
4502         case RT_LIMITPRIV:
4503                 if ((err = zonecfg_set_limitpriv(handle, prop_id)) != Z_OK)
4504                         zone_perror(zone, err, B_TRUE);
4505                 else
4506                         need_to_commit = B_TRUE;
4507                 return;
4508         case RT_BOOTARGS:
4509                 if ((err = zonecfg_set_bootargs(handle, prop_id)) != Z_OK)
4510                         zone_perror(zone, err, B_TRUE);
4511                 else
4512                         need_to_commit = B_TRUE;
4513                 return;
4514         case RT_SCHED:
4515                 if ((err = zonecfg_set_sched(handle, prop_id)) != Z_OK)
4516                         zone_perror(zone, err, B_TRUE);
4517                 else
4518                         need_to_commit = B_TRUE;
4519                 return;
4520         case RT_IPTYPE:
4521                 if (strcmp(prop_id, "shared") == 0) {
4522                         iptype = ZS_SHARED;
4523                 } else if (strcmp(prop_id, "exclusive") == 0) {
4524                         iptype = ZS_EXCLUSIVE;
4525                 } else {
4526                         zerr(gettext("%s value must be '%s' or '%s'."),
4527                             pt_to_str(PT_IPTYPE), "shared", "exclusive");
```



```
4528                         saw_error = B_TRUE;
4529                         return;
4530                 }
4531                 if (iptype == ZS_EXCLUSIVE && !allow_exclusive()) {
4532                         saw_error = B_TRUE;
4533                         return;
4534                 }
4535                 if ((err = zonecfg_set_iptype(handle, iptype)) != Z_OK)
4536                         zone_perror(zone, err, B_TRUE);
4537                 else
4538                         need_to_commit = B_TRUE;
4539                 return;
4540         case RT_MAXLWPS:
4541                 set_aliased_rctl(ALIAS_MAXLWPS, prop_type, prop_id);
4542                 return;
4543         case RT_MAXPROCS:
4544                 set_aliased_rctl(ALIAS_MAXPROCS, prop_type, prop_id);
4545                 return;
4546         case RT_MAXSHMMEM:
4547                 set_aliased_rctl(ALIAS_MAXSHMMEM, prop_type, prop_id);
4548                 return;
4549         case RT_MAXSHMIDS:
4550                 set_aliased_rctl(ALIAS_MAXSHMIDS, prop_type, prop_id);
4551                 return;
4552         case RT_MAXMSGIDS:
4553                 set_aliased_rctl(ALIAS_MAXMSGIDS, prop_type, prop_id);
4554                 return;
4555         case RT_MAXSEMIDS:
4556                 set_aliased_rctl(ALIAS_MAXSEMIDS, prop_type, prop_id);
4557                 return;
4558         case RT_SHARES:
4559                 set_aliased_rctl(ALIAS_SHARES, prop_type, prop_id);
4560                 return;
4561         case RT_HOSTID:
4562                 if ((err = zonecfg_set_hostid(handle, prop_id)) != Z_OK) {
4563                         if (err == Z_TOO_BIG) {
4564                                 zerr(gettext("hostid string is too large: %s"),
4565                                     prop_id);
4566                                 saw_error = B_TRUE;
4567                         } else {
4568                                 zone_perror(pt_to_str(prop_type), err, B_TRUE);
4569                         }
4570                         return;
4571                 }
4572                 need_to_commit = B_TRUE;
4573                 return;
4574         case RT_FS_ALLOWED:
4575                 if ((err = zonecfg_set_fs_allowed(handle, prop_id)) != Z_OK)
4576                         zone_perror(zone, err, B_TRUE);
4577                 else
4578                         need_to_commit = B_TRUE;
4579                 return;
4580         case RT_FS:
4581                 switch (prop_type) {
4582                 case PT_DIR:
4583                         (void) strlcpy(in_progress_fstab.zone_fs_dir, prop_id,
4584                             sizeof (in_progress_fstab.zone_fs_dir));
4585                         return;
4586                 case PT_SPECIAL:
4587                         (void) strlcpy(in_progress_fstab.zone_fs_special,
4588                             prop_id,
4589                             sizeof (in_progress_fstab.zone_fs_special));
4590                         return;
4591                 case PT_RAW:
4592                         (void) strlcpy(in_progress_fstab.zone_fs_raw,
4593                             prop_id, sizeof (in_progress_fstab.zone_fs_raw));
```

```
4594                        return;
4595                case PT_TYPE:
4596                        if (!valid_fs_type(prop_id)) {
4597                                zerr(gettext("\"%s\" is not a valid %s."),
4598                                    prop_id, pt_to_str(PT_TYPE));
4599                                saw_error = B_TRUE;
4600                                return;
4601                        }
4602                        (void) strlcpy(in_progress_fstab.zone_fs_type, prop_id,
4603                            sizeof (in_progress_fstab.zone_fs_type));
4604                        return;
4605                case PT_OPTIONS:
4606                        if (pp->pv_type != PROP_VAL_SIMPLE &&
4607                            pp->pv_type != PROP_VAL_LIST) {
4608                                zerr(gettext("A %s or %s value was expected "
4609                                    "here."), pvt_to_str(PROP_VAL_SIMPLE),
4610                                    pvt_to_str(PROP_VAL_LIST));
4611                                saw_error = B_TRUE;
4612                                return;
4613                        }
4614                        zonecfg_free_fs_option_list(
4615                            in_progress_fstab.zone_fs_options);
4616                        in_progress_fstab.zone_fs_options = NULL;
4617                        if (!(pp->pv_type == PROP_VAL_LIST &&
4618                            pp->pv_list == NULL))
4619                                add_property(cmd);
4620                        return;
4621                default:
4622                        break;
4623                }
4624                zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE, B_TRUE);
4625                long_usage(CMD_SET, B_TRUE);
4626                usage(B_FALSE, HELP_PROPS);
4627                return;
4628        case RT_NET:
4629                switch (prop_type) {
4630                case PT_ADDRESS:
4631                case PT_ALLOWED_ADDRESS:
4632                        if (validate_net_address_syntax(prop_id, B_FALSE)
4633                            != Z_OK) {
4634                                saw_error = B_TRUE;
4635                                return;
4636                        }
4637                        set_in_progress_nwiftab_address(prop_id, prop_type);
4638                        break;
4639                case PT_PHYSICAL:
4640                        if (validate_net_physical_syntax(prop_id) != Z_OK) {
4641                                saw_error = B_TRUE;
4642                                return;
4643                        }
4644                        (void) strlcpy(in_progress_nwiftab.zone_nwif_physical,
4645                            prop_id,
4646                            sizeof (in_progress_nwiftab.zone_nwif_physical));
4647                        break;
4648                case PT_DEFROUTER:
4649                        if (validate_net_address_syntax(prop_id, B_TRUE)
4650                            != Z_OK) {
4651                                saw_error = B_TRUE;
4652                                return;
4653                        }
4654                        (void) strlcpy(in_progress_nwiftab.zone_nwif_defrouter,
4655                            prop_id,
4656                            sizeof (in_progress_nwiftab.zone_nwif_defrouter));
4657                        break;
4658                default:
4659                        zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
```



```
4660                            B_TRUE);
4661                        long_usage(CMD_SET, B_TRUE);
4662                        usage(B_FALSE, HELP_PROPS);
4663                        return;
4664                }
4665                return;
4666        case RT_DEVICE:
4667                switch (prop_type) {
4668                case PT_MATCH:
4669                        (void) strlcpy(in_progress_devtab.zone_dev_match,
4670                            prop_id,
4671                            sizeof (in_progress_devtab.zone_dev_match));
4672                        break;
4673                default:
4674                        zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
4675                            B_TRUE);
4676                        long_usage(CMD_SET, B_TRUE);
4677                        usage(B_FALSE, HELP_PROPS);
4678                        return;
4679                }
4680                return;
4681        case RT_RCTL:
4682                switch (prop_type) {
4683                case PT_NAME:
4684                        if (!zonecfg_valid_rctlname(prop_id)) {
4685                                zerr(gettext("'%s' is not a valid zone %s "
4686                                    "name."), prop_id, rt_to_str(RT_RCTL));
4687                                return;
4688                        }
4689                        (void) strlcpy(in_progress_rctltab.zone_rctl_name,
4690                            prop_id,
4691                            sizeof (in_progress_rctltab.zone_rctl_name));
4692                        break;
4693                case PT_VALUE:
4694                        if (pp->pv_type != PROP_VAL_COMPLEX &&
4695                            pp->pv_type != PROP_VAL_LIST) {
4696                                zerr(gettext("A %s or %s value was expected "
4697                                    "here."), pvt_to_str(PROP_VAL_COMPLEX),
4698                                    pvt_to_str(PROP_VAL_LIST));
4699                                saw_error = B_TRUE;
4700                                return;
4701                        }
4702                        zonecfg_free_rctl_value_list(
4703                            in_progress_rctltab.zone_rctl_valptr);
4704                        in_progress_rctltab.zone_rctl_valptr = NULL;
4705                        if (!(pp->pv_type == PROP_VAL_LIST &&
4706                            pp->pv_list == NULL))
4707                                add_property(cmd);
4708                        break;
4709                default:
4710                        zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
4711                            B_TRUE);
4712                        long_usage(CMD_SET, B_TRUE);
4713                        usage(B_FALSE, HELP_PROPS);
4714                        return;
4715                }
4716                return;
4717        case RT_ATTR:
4718                switch (prop_type) {
4719                case PT_NAME:
4720                        (void) strlcpy(in_progress_attrtab.zone_attr_name,
4721                            prop_id,
4722                            sizeof (in_progress_attrtab.zone_attr_name));
4723                        break;
4724                case PT_TYPE:
4725                        (void) strlcpy(in_progress_attrtab.zone_attr_type,
```

```
4726                            prop_id,
4727                            sizeof (in_progress_attrtab.zone_attr_type));
4728                    break;
4729            case PT_VALUE:
4730                    (void) strlcpy(in_progress_attrtab.zone_attr_value,
4731                            prop_id,
4732                            sizeof (in_progress_attrtab.zone_attr_value));
4733                    break;
4734            default:
4735                    zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
4736                            B_TRUE);
4737                    long_usage(CMD_SET, B_TRUE);
4738                    usage(B_FALSE, HELP_PROPS);
4739                    return;
4740            }
4741            return;
4742    case RT_DATASET:
4743            switch (prop_type) {
4744            case PT_NAME:
4745                    (void) strlcpy(in_progress_dstab.zone_dataset_name,
4746                            prop_id,
4747                            sizeof (in_progress_dstab.zone_dataset_name));
4748                    return;
4749            default:
4750                    break;
4751            }
4752            zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE, B_TRUE);
4753            long_usage(CMD_SET, B_TRUE);
4754            usage(B_FALSE, HELP_PROPS);
4755            return;
4756    case RT_DCPU:
4757            switch (prop_type) {
4758            char *lowp, *highp;

4760            case PT_NCPUS:
4761                    lowp = prop_id;
4762                    if ((highp = strchr(prop_id, '-')) != NULL)
4763                            *highp++ = '\0';
4764                    else
4765                            highp = lowp;

4767                    /* Make sure the input makes sense. */
4768                    if (!zonecfg_valid_ncpus(lowp, highp)) {
4769                            zerr(gettext("%s property is out of range."),
4770                                pt_to_str(PT_NCPUS));
4771                            saw_error = B_TRUE;
4772                            return;
4773                    }

4775                    (void) strlcpy(
4776                        in_progress_psettab.zone_ncpu_min, lowp,
4777                        sizeof (in_progress_psettab.zone_ncpu_min));
4778                    (void) strlcpy(
4779                        in_progress_psettab.zone_ncpu_max, highp,
4780                        sizeof (in_progress_psettab.zone_ncpu_max));
4781                    return;
4782            case PT_IMPORTANCE:
4783                    /* Make sure the value makes sense. */
4784                    if (!zonecfg_valid_importance(prop_id)) {
4785                            zerr(gettext("%s property is out of range."),
4786                                pt_to_str(PT_IMPORTANCE));
4787                            saw_error = B_TRUE;
4788                            return;
4789                    }

4791                    (void) strlcpy(in_progress_psettab.zone_importance,
```



```
4792                            prop_id,
4793                            sizeof (in_progress_psettab.zone_importance));
4794                    return;
4795            default:
4796                    break;
4797            }
4798            zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE, B_TRUE);
4799            long_usage(CMD_SET, B_TRUE);
4800            usage(B_FALSE, HELP_PROPS);
4801            return;
4802    case RT_PCAP:
4803            if (prop_type != PT_NCPUS) {
4804                    zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
4805                        B_TRUE);
4806                    long_usage(CMD_SET, B_TRUE);
4807                    usage(B_FALSE, HELP_PROPS);
4808                    return;
4809            }

4811            /*
4812             * We already checked that an rctl alias is allowed in
4813             * the add_resource() function.
4814             */

4816            if ((cap = strtof(prop_id, &unitp)) <= 0 || *unitp != '\0' ||
4817                (int)(cap * 100) < 1) {
4818                    zerr(gettext("%s property is out of range."),
4819                        pt_to_str(PT_NCPUS));
4820                    saw_error = B_TRUE;
4821                    return;
4822            }

4824            if ((err = zonecfg_set_aliased_rctl(handle, ALIAS_CPUCAP,
4825                (int)(cap * 100))) != Z_OK)
4826                    zone_perror(zone, err, B_TRUE);
4827            else
4828                    need_to_commit = B_TRUE;
4829            return;
4830    case RT_MCAP:
4831            switch (prop_type) {
4832            case PT_PHYSICAL:
4833                    if (!zonecfg_valid_memlimit(prop_id, &mem_cap)) {
4834                            zerr(gettext("A positive number with a "
4835                                "required scale suffix (K, M, G or T) was "
4836                                "expected here."));
4837                            saw_error = B_TRUE;
4838                    } else if (mem_cap < ONE_MB) {
4839                            zerr(gettext("%s value is too small.  It must "
4840                                "be at least 1M."), pt_to_str(PT_PHYSICAL));
4841                            saw_error = B_TRUE;
4842                    } else {
4843                            snprintf(in_progress_mcaptab.zone_physmem_cap,
4844                                physmem_size, "%llu", mem_cap);
4845                    }
4846                    break;
4847            case PT_SWAP:
4848                    /*
4849                     * We have to check if an rctl is allowed here since
4850                     * there might already be a rctl defined that blocks
4851                     * the alias.
4852                     */
4853                    if (!zonecfg_aliased_rctl_ok(handle, ALIAS_MAXSWAP)) {
4854                            zone_perror(pt_to_str(PT_MAXSWAP),
4855                                Z_ALIAS_DISALLOW, B_FALSE);
4856                            saw_error = B_TRUE;
4857                            return;
```

```
4858                        }
4859
4860                        if (global_zone)
4861                                mem_limit = ONE_MB * 100;
4862                        else
4863                                mem_limit = ONE_MB * 50;
4864
4865                        if (!zonecfg_valid_memlimit(prop_id, &mem_cap)) {
4866                                zerr(gettext("A positive number with a "
4867                                    "required scale suffix (K, M, G or T) was "
4868                                    "expected here."));
4869                                saw_error = B_TRUE;
4870                        } else if (mem_cap < mem_limit) {
4871                                char buf[128];
4872
4873                                (void) snprintf(buf, sizeof (buf), "%llu",
4874                                    mem_limit);
4875                                bytes_to_units(buf, buf, sizeof (buf));
4876                                zerr(gettext("%s value is too small.  It must "
4877                                    "be at least %s."), pt_to_str(PT_SWAP),
4878                                    buf);
4879                                saw_error = B_TRUE;
4880                        } else {
4881                                if ((err = zonecfg_set_aliased_rctl(handle,
4882                                    ALIAS_MAXSWAP, mem_cap)) != Z_OK)
4883                                        zone_perror(zone, err, B_TRUE);
4884                                else
4885                                        need_to_commit = B_TRUE;
4886                        }
4887                        break;
4888                case PT_LOCKED:
4889                        /*
4890                         * We have to check if an rctl is allowed here since
4891                         * there might already be a rctl defined that blocks
4892                         * the alias.
4893                         */
4894                        if (!zonecfg_aliased_rctl_ok(handle,
4895                            ALIAS_MAXLOCKEDMEM)) {
4896                                zone_perror(pt_to_str(PT_LOCKED),
4897                                    Z_ALIAS_DISALLOW, B_FALSE);
4898                                saw_error = B_TRUE;
4899                                return;
4900                        }
4901
4902                        if (!zonecfg_valid_memlimit(prop_id, &mem_cap)) {
4903                                zerr(gettext("A non-negative number with a "
4904                                    "required scale suffix (K, M, G or T) was "
4905                                    "expected\nhere."));
4906                                saw_error = B_TRUE;
4907                        } else {
4908                                if ((err = zonecfg_set_aliased_rctl(handle,
4909                                    ALIAS_MAXLOCKEDMEM, mem_cap)) != Z_OK)
4910                                        zone_perror(zone, err, B_TRUE);
4911                                else
4912                                        need_to_commit = B_TRUE;
4913                        }
4914                        break;
4915                default:
4916                        zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
4917                            B_TRUE);
4918                        long_usage(CMD_SET, B_TRUE);
4919                        usage(B_FALSE, HELP_PROPS);
4920                        return;
4921                }
4922                return;
4923        case RT_ADMIN:
```



```
4924                switch (prop_type) {
4925                case PT_USER:
4926                        (void) strlcpy(in_progress_admintab.zone_admin_user,
4927                            prop_id,
4928                            sizeof (in_progress_admintab.zone_admin_user));
4929                        return;
4930                case PT_AUTHS:
4931                        (void) strlcpy(in_progress_admintab.zone_admin_auths,
4932                            prop_id,
4933                            sizeof (in_progress_admintab.zone_admin_auths));
4934                        return;
4935                default:
4936                        zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
4937                            B_TRUE);
4938                        long_usage(CMD_SET, B_TRUE);
4939                        usage(B_FALSE, HELP_PROPS);
4940                        return;
4941                }
4942        case RT_SECFLAGS: {
4943                char *propstr;
4944
4945                switch (prop_type) {
4946                case PT_DEFAULT:
4947                        propstr = in_progress_secflagstab.zone_secflags_default;
4948                        break;
4949                case PT_UPPER:
4950                        propstr = in_progress_secflagstab.zone_secflags_upper;
4951                        break;
4952                case PT_LOWER:
4953                        propstr = in_progress_secflagstab.zone_secflags_lower;
4954                        break;
4955                default:
4956                        zone_perror(pt_to_str(prop_type), Z_NO_PROPERTY_TYPE,
4957                            B_TRUE);
4958                        long_usage(CMD_SET, B_TRUE);
4959                        usage(B_FALSE, HELP_PROPS);
4960                        return;
4961                }
4962                (void) strlcpy(propstr, prop_id, ZONECFG_SECFLAGS_MAX);
4963                return;
4964        }
4965        default:
4966                zone_perror(rt_to_str(res_type), Z_NO_RESOURCE_TYPE, B_TRUE);
4967                long_usage(CMD_SET, B_TRUE);
4968                usage(B_FALSE, HELP_RESOURCES);
4969                return;
4970        }
4971 }
4972
4973 static void
4974 output_prop(FILE *fp, int pnum, char *pval, boolean_t print_notspec)
4975 {
4976        char *qstr;
4977
4978        if (*pval != '\0') {
4979                qstr = quoteit(pval);
4980                if (pnum == PT_SWAP || pnum == PT_LOCKED)
4981                        (void) fprintf(fp, "\t[%s: %s]\n", pt_to_str(pnum),
4982                            qstr);
4983                else
4984                        (void) fprintf(fp, "\t%s: %s\n", pt_to_str(pnum), qstr);
4985                free(qstr);
4986        } else if (print_notspec)
4987                (void) fprintf(fp, gettext("\t%s not specified\n"),
4988                    pt_to_str(pnum));
4989 }
```

```
4991 static void
4992 info_zonename(zone_dochandle_t handle, FILE *fp)
4993 {
4994         char zonename[ZONENAME_MAX];

4996         if (zonecfg_get_name(handle, zonename, sizeof (zonename)) == Z_OK)
4997                 (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_ZONENAME),
4998                     zonename);
4999         else
5000                 (void) fprintf(fp, gettext("%s not specified\n"),
5001                     pt_to_str(PT_ZONENAME));
5002 }

5004 static void
5005 info_zonepath(zone_dochandle_t handle, FILE *fp)
5006 {
5007         char zonepath[MAXPATHLEN];

5009         if (zonecfg_get_zonepath(handle, zonepath, sizeof (zonepath)) == Z_OK)
5010                 (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_ZONEPATH),
5011                     zonepath);
5012         else {
5013                 (void) fprintf(fp, gettext("%s not specified\n"),
5014                     pt_to_str(PT_ZONEPATH));
5015         }
5016 }

5018 static void
5019 info_brand(zone_dochandle_t handle, FILE *fp)
5020 {
5021         char brand[MAXNAMELEN];

5023         if (zonecfg_get_brand(handle, brand, sizeof (brand)) == Z_OK)
5024                 (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_BRAND),
5025                     brand);
5026         else
5027                 (void) fprintf(fp, "%s %s\n", pt_to_str(PT_BRAND),
5028                     gettext("not specified"));
5029 }

5031 static void
5032 info_autoboot(zone_dochandle_t handle, FILE *fp)
5033 {
5034         boolean_t autoboot;
5035         int err;

5037         if ((err = zonecfg_get_autoboot(handle, &autoboot)) == Z_OK)
5038                 (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_AUTOBOOT),
5039                     autoboot ? "true" : "false");
5040         else
5041                 zone_perror(zone, err, B_TRUE);
5042 }

5044 static void
5045 info_pool(zone_dochandle_t handle, FILE *fp)
5046 {
5047         char pool[MAXNAMELEN];
5048         int err;

5050         if ((err = zonecfg_get_pool(handle, pool, sizeof (pool))) == Z_OK)
5051                 (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_POOL), pool);
5052         else
5053                 zone_perror(zone, err, B_TRUE);
5054 }
```



```
5056 static void
5057 info_limitpriv(zone_dochandle_t handle, FILE *fp)
5058 {
5059         char *limitpriv;
5060         int err;

5062         if ((err = zonecfg_get_limitpriv(handle, &limitpriv)) == Z_OK) {
5063                 (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_LIMITPRIV),
5064                     limitpriv);
5065                 free(limitpriv);
5066         } else {
5067                 zone_perror(zone, err, B_TRUE);
5068         }
5069 }

5071 static void
5072 info_bootargs(zone_dochandle_t handle, FILE *fp)
5073 {
5074         char bootargs[BOOTARGS_MAX];
5075         int err;

5077         if ((err = zonecfg_get_bootargs(handle, bootargs,
5078             sizeof (bootargs))) == Z_OK) {
5079                 (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_BOOTARGS),
5080                     bootargs);
5081         } else {
5082                 zone_perror(zone, err, B_TRUE);
5083         }
5084 }

5086 static void
5087 info_sched(zone_dochandle_t handle, FILE *fp)
5088 {
5089         char sched[MAXNAMELEN];
5090         int err;

5092         if ((err = zonecfg_get_sched_class(handle, sched, sizeof (sched)))
5093             == Z_OK) {
5094                 (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_SCHED), sched);
5095         } else {
5096                 zone_perror(zone, err, B_TRUE);
5097         }
5098 }

5100 static void
5101 info_iptype(zone_dochandle_t handle, FILE *fp)
5102 {
5103         zone_iptype_t iptype;
5104         int err;

5106         if ((err = zonecfg_get_iptype(handle, &iptype)) == Z_OK) {
5107                 switch (iptype) {
5108                 case ZS_SHARED:
5109                         (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_IPTYPE),
5110                             "shared");
5111                         break;
5112                 case ZS_EXCLUSIVE:
5113                         (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_IPTYPE),
5114                             "exclusive");
5115                         break;
5116                 }
5117         } else {
5118                 zone_perror(zone, err, B_TRUE);
5119         }
5120 }
```

```
5122 static void
5123 info_hostid(zone_dochandle_t handle, FILE *fp)
5124 {
5125         char hostidp[HW_HOSTID_LEN];
5126         int err;
5127
5128         if ((err = zonecfg_get_hostid(handle, hostidp,
5129             sizeof (hostidp))) == Z_OK) {
5130                 (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_HOSTID), hostidp);
5131         } else if (err == Z_BAD_PROPERTY) {
5132                 (void) fprintf(fp, "%s: \n", pt_to_str(PT_HOSTID));
5133         } else {
5134                 zone_perror(zone, err, B_TRUE);
5135         }
5136 }
5137
5138 static void
5139 info_fs_allowed(zone_dochandle_t handle, FILE *fp)
5140 {
5141         char fsallowedp[ZONE_FS_ALLOWED_MAX];
5142         int err;
5143
5144         if ((err = zonecfg_get_fs_allowed(handle, fsallowedp,
5145             sizeof (fsallowedp))) == Z_OK) {
5146                 (void) fprintf(fp, "%s: %s\n", pt_to_str(PT_FS_ALLOWED),
5147                     fsallowedp);
5148         } else if (err == Z_BAD_PROPERTY) {
5149                 (void) fprintf(fp, "%s: \n", pt_to_str(PT_FS_ALLOWED));
5150         } else {
5151                 zone_perror(zone, err, B_TRUE);
5152         }
5153 }
5154
5155 static void
5156 output_fs(FILE *fp, struct zone_fstab *fstab)
5157 {
5158         zone_fsopt_t *this;
5159
5160         (void) fprintf(fp, "%s:\n", rt_to_str(RT_FS));
5161         output_prop(fp, PT_DIR, fstab->zone_fs_dir, B_TRUE);
5162         output_prop(fp, PT_SPECIAL, fstab->zone_fs_special, B_TRUE);
5163         output_prop(fp, PT_RAW, fstab->zone_fs_raw, B_TRUE);
5164         output_prop(fp, PT_TYPE, fstab->zone_fs_type, B_TRUE);
5165         (void) fprintf(fp, "\t%s: [", pt_to_str(PT_OPTIONS));
5166         for (this = fstab->zone_fs_options; this != NULL;
5167             this = this->zone_fsopt_next) {
5168                 if (strchr(this->zone_fsopt_opt, '=’))
5169                         (void) fprintf(fp, "\"%s\"", this->zone_fsopt_opt);
5170                 else
5171                         (void) fprintf(fp, "%s", this->zone_fsopt_opt);
5172                 if (this->zone_fsopt_next != NULL)
5173                         (void) fprintf(fp, ",");
5174         }
5175         (void) fprintf(fp, "]\n");
5176 }
5177
5178 static void
5179 info_fs(zone_dochandle_t handle, FILE *fp, cmd_t *cmd)
5180 {
5181         struct zone_fstab lookup, user;
5182         boolean_t output = B_FALSE;
5183
5184         if (zonecfg_setfsent(handle) != Z_OK)
5185                 return;
5186         while (zonecfg_getfsent(handle, &lookup) == Z_OK) {
5187                 if (cmd->cmd_prop_nv_pairs == 0) {
```



```
5188                         output_fs(fp, &lookup);
5189                         goto loopend;
5190                 }
5191                 if (fill_in_fstab(cmd, &user, B_TRUE) != Z_OK)
5192                         goto loopend;
5193                 if (strlen(user.zone_fs_dir) > 0 &&
5194                     strcmp(user.zone_fs_dir, lookup.zone_fs_dir) != 0)
5195                         goto loopend;   /* no match */
5196                 if (strlen(user.zone_fs_special) > 0 &&
5197                     strcmp(user.zone_fs_special, lookup.zone_fs_special) != 0)
5198                         goto loopend;   /* no match */
5199                 if (strlen(user.zone_fs_type) > 0 &&
5200                     strcmp(user.zone_fs_type, lookup.zone_fs_type) != 0)
5201                         goto loopend;   /* no match */
5202                 output_fs(fp, &lookup);
5203                 output = B_TRUE;
5204 loopend:
5205                 zonecfg_free_fs_option_list(lookup.zone_fs_options);
5206         }
5207         (void) zonecfg_endfsent(handle);
5208         /*
5209          * If a property n/v pair was specified, warn the user if there was
5210          * nothing to output.
5211          */
5212         if (!output && cmd->cmd_prop_nv_pairs > 0)
5213                 (void) printf(gettext("No such %s resource.\n"),
5214                     rt_to_str(RT_FS));
5215 }
5216
5217 static void
5218 output_net(FILE *fp, struct zone_nwiftab *nwiftab)
5219 {
5220         (void) fprintf(fp, "%s:\n", rt_to_str(RT_NET));
5221         output_prop(fp, PT_ADDRESS, nwiftab->zone_nwif_address, B_TRUE);
5222         output_prop(fp, PT_ALLOWED_ADDRESS,
5223             nwiftab->zone_nwif_allowed_address, B_TRUE);
5224         output_prop(fp, PT_PHYSICAL, nwiftab->zone_nwif_physical, B_TRUE);
5225         output_prop(fp, PT_DEFROUTER, nwiftab->zone_nwif_defrouter, B_TRUE);
5226 }
5227
5228 static void
5229 info_net(zone_dochandle_t handle, FILE *fp, cmd_t *cmd)
5230 {
5231         struct zone_nwiftab lookup, user;
5232         boolean_t output = B_FALSE;
5233
5234         if (zonecfg_setnwifent(handle) != Z_OK)
5235                 return;
5236         while (zonecfg_getnwifent(handle, &lookup) == Z_OK) {
5237                 if (cmd->cmd_prop_nv_pairs == 0) {
5238                         output_net(fp, &lookup);
5239                         continue;
5240                 }
5241                 if (fill_in_nwiftab(cmd, &user, B_TRUE) != Z_OK)
5242                         continue;
5243                 if (strlen(user.zone_nwif_physical) > 0 &&
5244                     strcmp(user.zone_nwif_physical,
5245                     lookup.zone_nwif_physical) != 0)
5246                         continue;       /* no match */
5247                 /* If present make sure it matches */
5248                 if (strlen(user.zone_nwif_address) > 0 &&
5249                     !zonecfg_same_net_address(user.zone_nwif_address,
5250                     lookup.zone_nwif_address))
5251                         continue;       /* no match */
5252                 output_net(fp, &lookup);
5253                 output = B_TRUE;
```

```
5254         }
5255         (void) zonecfg_endnwifent(handle);
5256         /*
5257          * If a property n/v pair was specified, warn the user if there was
5258          * nothing to output.
5259          */
5260         if (!output && cmd->cmd_prop_nv_pairs > 0)
5261                 (void) printf(gettext("No such %s resource.\n"),
5262                     rt_to_str(RT_NET));
5263 }

5265 static void
5266 output_dev(FILE *fp, struct zone_devtab *devtab)
5267 {
5268         (void) fprintf(fp, "%s:\n", rt_to_str(RT_DEVICE));
5269         output_prop(fp, PT_MATCH, devtab->zone_dev_match, B_TRUE);
5270 }

5272 static void
5273 info_dev(zone_dochandle_t handle, FILE *fp, cmd_t *cmd)
5274 {
5275         struct zone_devtab lookup, user;
5276         boolean_t output = B_FALSE;

5278         if (zonecfg_setdevent(handle) != Z_OK)
5279                 return;
5280         while (zonecfg_getdevent(handle, &lookup) == Z_OK) {
5281                 if (cmd->cmd_prop_nv_pairs == 0) {
5282                         output_dev(fp, &lookup);
5283                         continue;
5284                 }
5285                 if (fill_in_devtab(cmd, &user, B_TRUE) != Z_OK)
5286                         continue;
5287                 if (strlen(user.zone_dev_match) > 0 &&
5288                     strcmp(user.zone_dev_match, lookup.zone_dev_match) != 0)
5289                         continue;       /* no match */
5290                 output_dev(fp, &lookup);
5291                 output = B_TRUE;
5292         }
5293         (void) zonecfg_enddevent(handle);
5294         /*
5295          * If a property n/v pair was specified, warn the user if there was
5296          * nothing to output.
5297          */
5298         if (!output && cmd->cmd_prop_nv_pairs > 0)
5299                 (void) printf(gettext("No such %s resource.\n"),
5300                     rt_to_str(RT_DEVICE));
5301 }

5303 static void
5304 output_rctl(FILE *fp, struct zone_rctltab *rctltab)
5305 {
5306         struct zone_rctlvaltab *valptr;

5308         (void) fprintf(fp, "%s:\n", rt_to_str(RT_RCTL));
5309         output_prop(fp, PT_NAME, rctltab->zone_rctl_name, B_TRUE);
5310         for (valptr = rctltab->zone_rctl_valptr; valptr != NULL;
5311             valptr = valptr->zone_rctlval_next) {
5312                 fprintf(fp, "\t%s: (%s=%s,%s=%s,%s=%s)\n",
5313                     pt_to_str(PT_VALUE),
5314                     pt_to_str(PT_PRIV), valptr->zone_rctlval_priv,
5315                     pt_to_str(PT_LIMIT), valptr->zone_rctlval_limit,
5316                     pt_to_str(PT_ACTION), valptr->zone_rctlval_action);
5317         }
5318 }
```



```
5320 static void
5321 info_rctl(zone_dochandle_t handle, FILE *fp, cmd_t *cmd)
5322 {
5323         struct zone_rctltab lookup, user;
5324         boolean_t output = B_FALSE;

5326         if (zonecfg_setrctlent(handle) != Z_OK)
5327                 return;
5328         while (zonecfg_getrctlent(handle, &lookup) == Z_OK) {
5329                 if (cmd->cmd_prop_nv_pairs == 0) {
5330                         output_rctl(fp, &lookup);
5331                 } else if (fill_in_rctltab(cmd, &user, B_TRUE) == Z_OK &&
5332                     (strlen(user.zone_rctl_name) == 0 ||
5333                     strcmp(user.zone_rctl_name, lookup.zone_rctl_name) == 0)) {
5334                         output_rctl(fp, &lookup);
5335                         output = B_TRUE;
5336                 }
5337                 zonecfg_free_rctl_value_list(lookup.zone_rctl_valptr);
5338         }
5339         (void) zonecfg_endrctlent(handle);
5340         /*
5341          * If a property n/v pair was specified, warn the user if there was
5342          * nothing to output.
5343          */
5344         if (!output && cmd->cmd_prop_nv_pairs > 0)
5345                 (void) printf(gettext("No such %s resource.\n"),
5346                     rt_to_str(RT_RCTL));
5347 }

5349 static void
5350 output_attr(FILE *fp, struct zone_attrtab *attrtab)
5351 {
5352         (void) fprintf(fp, "%s:\n", rt_to_str(RT_ATTR));
5353         output_prop(fp, PT_NAME, attrtab->zone_attr_name, B_TRUE);
5354         output_prop(fp, PT_TYPE, attrtab->zone_attr_type, B_TRUE);
5355         output_prop(fp, PT_VALUE, attrtab->zone_attr_value, B_TRUE);
5356 }

5358 static void
5359 info_attr(zone_dochandle_t handle, FILE *fp, cmd_t *cmd)
5360 {
5361         struct zone_attrtab lookup, user;
5362         boolean_t output = B_FALSE;

5364         if (zonecfg_setattrent(handle) != Z_OK)
5365                 return;
5366         while (zonecfg_getattrent(handle, &lookup) == Z_OK) {
5367                 if (cmd->cmd_prop_nv_pairs == 0) {
5368                         output_attr(fp, &lookup);
5369                         continue;
5370                 }
5371                 if (fill_in_attrtab(cmd, &user, B_TRUE) != Z_OK)
5372                         continue;
5373                 if (strlen(user.zone_attr_name) > 0 &&
5374                     strcmp(user.zone_attr_name, lookup.zone_attr_name) != 0)
5375                         continue;       /* no match */
5376                 if (strlen(user.zone_attr_type) > 0 &&
5377                     strcmp(user.zone_attr_type, lookup.zone_attr_type) != 0)
5378                         continue;       /* no match */
5379                 if (strlen(user.zone_attr_value) > 0 &&
5380                     strcmp(user.zone_attr_value, lookup.zone_attr_value) != 0)
5381                         continue;       /* no match */
5382                 output_attr(fp, &lookup);
5383                 output = B_TRUE;
5384         }
5385         (void) zonecfg_endattrent(handle);
```

```
5386         /*
5387          * If a property n/v pair was specified, warn the user if there was
5388          * nothing to output.
5389          */
5390         if (!output && cmd->cmd_prop_nv_pairs > 0)
5391                 (void) printf(gettext("No such %s resource.\n"),
5392                     rt_to_str(RT_ATTR));
5393 }

5395 static void
5396 output_ds(FILE *fp, struct zone_dstab *dstab)
5397 {
5398         (void) fprintf(fp, "%s:\n", rt_to_str(RT_DATASET));
5399         output_prop(fp, PT_NAME, dstab->zone_dataset_name, B_TRUE);
5400 }

5402 static void
5403 info_ds(zone_dochandle_t handle, FILE *fp, cmd_t *cmd)
5404 {
5405         struct zone_dstab lookup, user;
5406         boolean_t output = B_FALSE;

5408         if (zonecfg_setdsent(handle) != Z_OK)
5409                 return;
5410         while (zonecfg_getdsent(handle, &lookup) == Z_OK) {
5411                 if (cmd->cmd_prop_nv_pairs == 0) {
5412                         output_ds(fp, &lookup);
5413                         continue;
5414                 }
5415                 if (fill_in_dstab(cmd, &user, B_TRUE) != Z_OK)
5416                         continue;
5417                 if (strlen(user.zone_dataset_name) > 0 &&
5418                     strcmp(user.zone_dataset_name,
5419                     lookup.zone_dataset_name) != 0)
5420                         continue;       /* no match */
5421                 output_ds(fp, &lookup);
5422                 output = B_TRUE;
5423         }
5424         (void) zonecfg_enddsent(handle);
5425         /*
5426          * If a property n/v pair was specified, warn the user if there was
5427          * nothing to output.
5428          */
5429         if (!output && cmd->cmd_prop_nv_pairs > 0)
5430                 (void) printf(gettext("No such %s resource.\n"),
5431                     rt_to_str(RT_DATASET));
5432 }

5434 static void
5435 output_pset(FILE *fp, struct zone_psettab *psettab)
5436 {
5437         (void) fprintf(fp, "%s:\n", rt_to_str(RT_DCPU));
5438         if (strcmp(psettab->zone_ncpu_min, psettab->zone_ncpu_max) == 0)
5439                 (void) fprintf(fp, "\t%s: %s\n", pt_to_str(PT_NCPUS),
5440                     psettab->zone_ncpu_max);
5441         else
5442                 (void) fprintf(fp, "\t%s: %s-%s\n", pt_to_str(PT_NCPUS),
5443                     psettab->zone_ncpu_min, psettab->zone_ncpu_max);
5444         if (psettab->zone_importance[0] != '\0')
5445                 (void) fprintf(fp, "\t%s: %s\n", pt_to_str(PT_IMPORTANCE),
5446                     psettab->zone_importance);
5447 }

5449 static void
5450 info_pset(zone_dochandle_t handle, FILE *fp)
5451 {
```



```
5452         struct zone_psettab lookup;

5454         if (zonecfg_getpsetent(handle, &lookup) == Z_OK)
5455                 output_pset(fp, &lookup);
5456 }

5458 static void
5459 output_pcap(FILE *fp)
5460 {
5461         uint64_t cap;

5463         if (zonecfg_get_aliased_rctl(handle, ALIAS_CPUCAP, &cap) == Z_OK) {
5464                 float scaled = (float)cap / 100;
5465                 (void) fprintf(fp, "%s:\n", rt_to_str(RT_PCAP));
5466                 (void) fprintf(fp, "\t[%s: %.2f]\n", pt_to_str(PT_NCPUS),
5467                     scaled);
5468         }
5469 }

5471 static void
5472 info_pcap(FILE *fp)
5473 {
5474         output_pcap(fp);
5475 }


5478 static void
5479 info_aliased_rctl(zone_dochandle_t handle, FILE *fp, char *alias)
5480 {
5481         uint64_t limit;

5483         if (zonecfg_get_aliased_rctl(handle, alias, &limit) == Z_OK) {
5484                 /* convert memory based properties */
5485                 if (strcmp(alias, ALIAS_MAXSHMMEM) == 0) {
5486                         char buf[128];

5488                         (void) snprintf(buf, sizeof (buf), "%llu", limit);
5489                         bytes_to_units(buf, buf, sizeof (buf));
5490                         (void) fprintf(fp, "[%s: %s]\n", alias, buf);
5491                         return;
5492                 }

5494                 (void) fprintf(fp, "[%s: %llu]\n", alias, limit);
5495         }
5496 }

5498 static void
5499 bytes_to_units(char *str, char *buf, int bufsize)
5500 {
5501         unsigned long long num;
5502         unsigned long long save = 0;
5503         char *units = "BKMGT";
5504         char *up = units;

5506         num = strtoll(str, NULL, 10);

5508         if (num < 1024) {
5509                 (void) snprintf(buf, bufsize, "%llu", num);
5510                 return;
5511         }

5513         while ((num >= 1024) && (*up != 'T')) {
5514                 up++; /* next unit of measurement */
5515                 save = num;
5516                 num = (num + 512) >> 10;
5517         }
```

```
5519         /* check if we should output a fraction.  snprintf will round for us */
5520         if (save % 1024 != 0 && ((save >> 10) < 10))
5521                 (void) snprintf(buf, bufsize, "%2.1f%c", ((float)save / 1024),
5522                     *up);
5523         else
5524                 (void) snprintf(buf, bufsize, "%llu%c", num, *up);
5525 }

5527 static void
5528 output_mcap(FILE *fp, struct zone_mcaptab *mcaptab, int showswap,
5529     uint64_t maxswap, int showlocked, uint64_t maxlocked)
5530 {
5531         char buf[128];

5533         (void) fprintf(fp, "%s:\n", rt_to_str(RT_MCAP));
5534         if (mcaptab->zone_physmem_cap[0] != '\0') {
5535                 bytes_to_units(mcaptab->zone_physmem_cap, buf, sizeof (buf));
5536                 output_prop(fp, PT_PHYSICAL, buf, B_TRUE);
5537         }

5539         if (showswap == Z_OK) {
5540                 (void) snprintf(buf, sizeof (buf), "%llu", maxswap);
5541                 bytes_to_units(buf, buf, sizeof (buf));
5542                 output_prop(fp, PT_SWAP, buf, B_TRUE);
5543         }

5545         if (showlocked == Z_OK) {
5546                 (void) snprintf(buf, sizeof (buf), "%llu", maxlocked);
5547                 bytes_to_units(buf, buf, sizeof (buf));
5548                 output_prop(fp, PT_LOCKED, buf, B_TRUE);
5549         }
5550 }

5552 static void
5553 info_mcap(zone_dochandle_t handle, FILE *fp)
5554 {
5555         int res1, res2, res3;
5556         uint64_t swap_limit;
5557         uint64_t locked_limit;
5558         struct zone_mcaptab lookup;

5560         bzero(&lookup, sizeof (lookup));
5561         res1 = zonecfg_getmcapent(handle, &lookup);
5562         res2 = zonecfg_get_aliased_rctl(handle, ALIAS_MAXSWAP, &swap_limit);
5563         res3 = zonecfg_get_aliased_rctl(handle, ALIAS_MAXLOCKEDMEM,
5564             &locked_limit);

5566         if (res1 == Z_OK || res2 == Z_OK || res3 == Z_OK)
5567                 output_mcap(fp, &lookup, res2, swap_limit, res3, locked_limit);
5568 }

5570 static void
5571 output_auth(FILE *fp, struct zone_admintab *admintab)
5572 {
5573         (void) fprintf(fp, "%s:\n", rt_to_str(RT_ADMIN));
5574         output_prop(fp, PT_USER, admintab->zone_admin_user, B_TRUE);
5575         output_prop(fp, PT_AUTHS, admintab->zone_admin_auths, B_TRUE);
5576 }

5578 static void
5579 output_secflags(FILE *fp, struct zone_secflagstab *sftab)
5580 {
5581         (void) fprintf(fp, "%s:\n", rt_to_str(RT_SECFLAGS));
5582         output_prop(fp, PT_DEFAULT, sftab->zone_secflags_default, B_TRUE);
5583         output_prop(fp, PT_LOWER, sftab->zone_secflags_lower, B_TRUE);
```



```
5584         output_prop(fp, PT_UPPER, sftab->zone_secflags_upper, B_TRUE);
5585 }

5587 static void
5588 info_auth(zone_dochandle_t handle, FILE *fp, cmd_t *cmd)
5589 {
5590         struct zone_admintab lookup, user;
5591         boolean_t output = B_FALSE;
5592         int err;

5594         if ((err = zonecfg_setadminent(handle)) != Z_OK) {
5595                 zone_perror(zone, err, B_TRUE);
5596                 return;
5597         }
5598         while (zonecfg_getadminent(handle, &lookup) == Z_OK) {
5599                 if (cmd->cmd_prop_nv_pairs == 0) {
5600                         output_auth(fp, &lookup);
5601                         continue;
5602                 }
5603                 if (fill_in_admintab(cmd, &user, B_TRUE) != Z_OK)
5604                         continue;
5605                 if (strlen(user.zone_admin_user) > 0 &&
5606                     strcmp(user.zone_admin_user, lookup.zone_admin_user) != 0)
5607                         continue;       /* no match */
5608                 output_auth(fp, &lookup);
5609                 output = B_TRUE;
5610         }
5611         (void) zonecfg_endadminent(handle);
5612         /*
5613          * If a property n/v pair was specified, warn the user if there was
5614          * nothing to output.
5615          */
5616         if (!output && cmd->cmd_prop_nv_pairs > 0)
5617                 (void) printf(gettext("No such %s resource.\n"),
5618                     rt_to_str(RT_ADMIN));
5619 }

5621 static void
5622 info_secflags(zone_dochandle_t handle, FILE *fp)
5623 {
5624         struct zone_secflagstab sftab;
2131         int err;

5626         if (zonecfg_lookup_secflags(handle, &sftab) == Z_OK) {
5627                 output_secflags(fp, &sftab);
2133         if ((err = zonecfg_lookup_secflags(handle, &sftab)) != Z_OK) {
2134                 zone_perror(zone, err, B_TRUE);
2135                 return;
5628         }

2138         output_secflags(fp, &sftab);
5629 }
```

_____***unchanged_portion_omitted_***

```
6074 /*
6075  * See the DTD for which attributes are required for which resources.
6076  *
6077  * This function can be called by commit_func(), which needs to save things,
6078  * in addition to the general call from parse_and_run(), which doesn't need
6079  * things saved.  Since the parameters are standardized, we distinguish by
6080  * having commit_func() call here with cmd->cmd_arg set to "save" to indicate
6081  * that a save is needed.
6082  */
6083 void
6084 verify_func(cmd_t *cmd)
6085 {
```

```
6086         struct zone_nwiftab nwiftab;
6087         struct zone_fstab fstab;
6088         struct zone_attrtab attrtab;
6089         struct zone_rctltab rctltab;
6090         struct zone_dstab dstab;
6091         struct zone_psettab psettab;
6092         struct zone_admintab admintab;
6093         struct zone_secflagstab secflagstab;
6094         char zonepath[MAXPATHLEN];
6095         char sched[MAXNAMELEN];
6096         char brand[MAXNAMELEN];
6097         char hostidp[HW_HOSTID_LEN];
6098         char fsallowedp[ZONE_FS_ALLOWED_MAX];
6099         priv_set_t *privs;
6100         char *privname = NULL;
6101         int err, ret_val = Z_OK, arg;
6102         int pset_res;
6103         boolean_t save = B_FALSE;
6104         boolean_t arg_err = B_FALSE;
6105         zone_iptype_t iptype;
6106         boolean_t has_cpu_shares = B_FALSE;
6107         boolean_t has_cpu_cap = B_FALSE;
6108         struct xif *tmp;

6110         optind = 0;
6111         while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "?")) != EOF) {
6112                 switch (arg) {
6113                 case '?':
6114                         longer_usage(CMD_VERIFY);
6115                         arg_err = B_TRUE;
6116                         break;
6117                 default:
6118                         short_usage(CMD_VERIFY);
6119                         arg_err = B_TRUE;
6120                         break;
6121                 }
6122         }
6123         if (arg_err)
6124                 return;

6126         if (optind > cmd->cmd_argc) {
6127                 short_usage(CMD_VERIFY);
6128                 return;
6129         }

6131         if (zone_is_read_only(CMD_VERIFY))
6132                 return;

6134         assert(cmd != NULL);

6136         if (cmd->cmd_argc > 0 && (strcmp(cmd->cmd_argv[0], "save") == 0))
6137                 save = B_TRUE;
6138         if (initialize(B_TRUE) != Z_OK)
6139                 return;

6141         if (zonecfg_get_zonepath(handle, zonepath, sizeof (zonepath)) != Z_OK &&
6142             !global_zone) {
6143                 zerr(gettext("%s not specified"), pt_to_str(PT_ZONEPATH));
6144                 ret_val = Z_REQD_RESOURCE_MISSING;
6145                 saw_error = B_TRUE;
6146         }
6147         if (strlen(zonepath) == 0 && !global_zone) {
6148                 zerr(gettext("%s cannot be empty."), pt_to_str(PT_ZONEPATH));
6149                 ret_val = Z_REQD_RESOURCE_MISSING;
6150                 saw_error = B_TRUE;
6151         }
```



```
6153         if ((err = zonecfg_get_brand(handle, brand, sizeof (brand))) != Z_OK) {
6154                 zone_perror(zone, err, B_TRUE);
6155                 return;
6156         }
6157         if ((err = brand_verify(handle)) != Z_OK) {
6158                 zone_perror(zone, err, B_TRUE);
6159                 return;
6160         }

6162         if (zonecfg_get_iptype(handle, &iptype) != Z_OK) {
6163                 zerr("%s %s", gettext("cannot get"), pt_to_str(PT_IPTYPE));
6164                 ret_val = Z_REQD_RESOURCE_MISSING;
6165                 saw_error = B_TRUE;
6166         }

6168         if ((privs = priv_allocset()) == NULL) {
6169                 zerr(gettext("%s: priv_allocset failed"), zone);
6170                 return;
6171         }
6172         if (zonecfg_get_privset(handle, privs, &privname) != Z_OK) {
6173                 zerr(gettext("%s: invalid privilege: %s"), zone, privname);
6174                 priv_freeset(privs);
6175                 free(privname);
6176                 return;
6177         }
6178         priv_freeset(privs);

6180         if (zonecfg_get_hostid(handle, hostidp,
6181             sizeof (hostidp)) == Z_INVALID_PROPERTY) {
6182                 zerr(gettext("%s: invalid hostid: %s"),
6183                     zone, hostidp);
6184                 return;
6185         }

6187         if (zonecfg_get_fs_allowed(handle, fsallowedp,
6188             sizeof (fsallowedp)) == Z_INVALID_PROPERTY) {
6189                 zerr(gettext("%s: invalid fs-allowed: %s"),
6190                     zone, fsallowedp);
6191                 return;
6192         }

6194         if ((err = zonecfg_setfsent(handle)) != Z_OK) {
6195                 zone_perror(zone, err, B_TRUE);
6196                 return;
6197         }
6198         while (zonecfg_getfsent(handle, &fstab) == Z_OK) {
6199                 check_reqd_prop(fstab.zone_fs_dir, RT_FS, PT_DIR, &ret_val);
6200                 check_reqd_prop(fstab.zone_fs_special, RT_FS, PT_SPECIAL,
6201                     &ret_val);
6202                 check_reqd_prop(fstab.zone_fs_type, RT_FS, PT_TYPE, &ret_val);

6204                 zonecfg_free_fs_option_list(fstab.zone_fs_options);
6205         }
6206         (void) zonecfg_endfsent(handle);

6208         if ((err = zonecfg_setnwifent(handle)) != Z_OK) {
6209                 zone_perror(zone, err, B_TRUE);
6210                 return;
6211         }
6212         while (zonecfg_getnwifent(handle, &nwiftab) == Z_OK) {
6213                 /*
6214                  * physical is required in all cases.
6215                  * A shared IP requires an address,
6216                  * and may include a default router, while
6217                  * an exclusive IP must have neither an address
```

```
6218                 * nor a default router.
6219                 * The physical interface name must be valid in all cases.
6220                 */
6221                check_reqd_prop(nwiftab.zone_nwif_physical, RT_NET,
6222                    PT_PHYSICAL, &ret_val);
6223                if (validate_net_physical_syntax(nwiftab.zone_nwif_physical) !=
6224                    Z_OK) {
6225                        saw_error = B_TRUE;
6226                        if (ret_val == Z_OK)
6227                                ret_val = Z_INVAL;
6228                }

6230                switch (iptype) {
6231                case ZS_SHARED:
6232                        check_reqd_prop(nwiftab.zone_nwif_address, RT_NET,
6233                            PT_ADDRESS, &ret_val);
6234                        if (strlen(nwiftab.zone_nwif_allowed_address) > 0) {
6235                                zerr(gettext("%s: %s cannot be specified "
6236                                    "for a shared IP type"),
6237                                    rt_to_str(RT_NET),
6238                                    pt_to_str(PT_ALLOWED_ADDRESS));
6239                                saw_error = B_TRUE;
6240                                if (ret_val == Z_OK)
6241                                        ret_val = Z_INVAL;
6242                        }
6243                        break;
6244                case ZS_EXCLUSIVE:
6245                        if (strlen(nwiftab.zone_nwif_address) > 0) {
6246                                zerr(gettext("%s: %s cannot be specified "
6247                                    "for an exclusive IP type"),
6248                                    rt_to_str(RT_NET), pt_to_str(PT_ADDRESS));
6249                                saw_error = B_TRUE;
6250                                if (ret_val == Z_OK)
6251                                        ret_val = Z_INVAL;
6252                        } else {
6253                                if (!add_nwif(&nwiftab)) {
6254                                        saw_error = B_TRUE;
6255                                        if (ret_val == Z_OK)
6256                                                ret_val = Z_INVAL;
6257                                }
6258                        }
6259                        break;
6260                }
6261        }
6262        for (tmp = xif; tmp != NULL; tmp = tmp->xif_next) {
6263                if (!tmp->xif_has_address && tmp->xif_has_defrouter) {
6264                        zerr(gettext("%s: %s for %s cannot be specified "
6265                            "without %s for an exclusive IP type"),
6266                            rt_to_str(RT_NET), pt_to_str(PT_DEFROUTER),
6267                            tmp->xif_name, pt_to_str(PT_ALLOWED_ADDRESS));
6268                        saw_error = B_TRUE;
6269                        ret_val = Z_INVAL;
6270                }
6271        }
6272        free(xif);
6273        xif = NULL;
6274        (void) zonecfg_endnwifent(handle);

6276        if ((err = zonecfg_setrctlent(handle)) != Z_OK) {
6277                zone_perror(zone, err, B_TRUE);
6278                return;
6279        }
6280        while (zonecfg_getrctlent(handle, &rctltab) == Z_OK) {
6281                check_reqd_prop(rctltab.zone_rctl_name, RT_RCTL, PT_NAME,
6282                    &ret_val);
```



```
6284                if (strcmp(rctltab.zone_rctl_name, "zone.cpu-shares") == 0)
6285                        has_cpu_shares = B_TRUE;

6287                if (strcmp(rctltab.zone_rctl_name, "zone.cpu-cap") == 0)
6288                        has_cpu_cap = B_TRUE;

6290                if (rctltab.zone_rctl_valptr == NULL) {
6291                        zerr(gettext("%s: no %s specified"),
6292                            rt_to_str(RT_RCTL), pt_to_str(PT_VALUE));
6293                        saw_error = B_TRUE;
6294                        if (ret_val == Z_OK)
6295                                ret_val = Z_REQD_PROPERTY_MISSING;
6296                } else {
6297                        zonecfg_free_rctl_value_list(rctltab.zone_rctl_valptr);
6298                }
6299        }
6300        (void) zonecfg_endrctlent(handle);

6302        if ((pset_res = zonecfg_lookup_pset(handle, &psettab)) == Z_OK &&
6303            has_cpu_shares) {
6304                zerr(gettext("%s zone.cpu-shares and %s are incompatible."),
6305                    rt_to_str(RT_RCTL), rt_to_str(RT_DCPU));
6306                saw_error = B_TRUE;
6307                if (ret_val == Z_OK)
6308                        ret_val = Z_INCOMPATIBLE;
6309        }

6311        if (has_cpu_shares && zonecfg_get_sched_class(handle, sched,
6312            sizeof (sched)) == Z_OK && strlen(sched) > 0 &&
6313            strcmp(sched, "FSS") != 0) {
6314                zerr(gettext("WARNING: %s zone.cpu-shares and %s=%s are "
6315                    "incompatible"),
6316                    rt_to_str(RT_RCTL), rt_to_str(RT_SCHED), sched);
6317                saw_error = B_TRUE;
6318                if (ret_val == Z_OK)
6319                        ret_val = Z_INCOMPATIBLE;
6320        }

6322        if (pset_res == Z_OK && has_cpu_cap) {
6323                zerr(gettext("%s zone.cpu-cap and the %s are incompatible."),
6324                    rt_to_str(RT_RCTL), rt_to_str(RT_DCPU));
6325                saw_error = B_TRUE;
6326                if (ret_val == Z_OK)
6327                        ret_val = Z_INCOMPATIBLE;
6328        }

6330        if ((err = zonecfg_setattrent(handle)) != Z_OK) {
6331                zone_perror(zone, err, B_TRUE);
6332                return;
6333        }
6334        while (zonecfg_getattrent(handle, &attrtab) == Z_OK) {
6335                check_reqd_prop(attrtab.zone_attr_name, RT_ATTR, PT_NAME,
6336                    &ret_val);
6337                check_reqd_prop(attrtab.zone_attr_type, RT_ATTR, PT_TYPE,
6338                    &ret_val);
6339                check_reqd_prop(attrtab.zone_attr_value, RT_ATTR, PT_VALUE,
6340                    &ret_val);
6341        }
6342        (void) zonecfg_endattrent(handle);

6344        if ((err = zonecfg_setdsent(handle)) != Z_OK) {
6345                zone_perror(zone, err, B_TRUE);
6346                return;
6347        }
6348        while (zonecfg_getdsent(handle, &dstab) == Z_OK) {
6349                if (strlen(dstab.zone_dataset_name) == 0) {
```

```
6350                        zerr("%s: %s %s", rt_to_str(RT_DATASET),
6351                            pt_to_str(PT_NAME), gettext("not specified"));
6352                        saw_error = B_TRUE;
6353                        if (ret_val == Z_OK)
6354                                ret_val = Z_REQD_PROPERTY_MISSING;
6355                } else if (!zfs_name_valid(dstab.zone_dataset_name,
6356                    ZFS_TYPE_FILESYSTEM)) {
6357                        zerr("%s: %s %s", rt_to_str(RT_DATASET),
6358                            pt_to_str(PT_NAME), gettext("invalid"));
6359                        saw_error = B_TRUE;
6360                        if (ret_val == Z_OK)
6361                                ret_val = Z_BAD_PROPERTY;
6362                }

6364        }
6365        (void) zonecfg_enddsent(handle);

6367        if ((err = zonecfg_setadminent(handle)) != Z_OK) {
6368                zone_perror(zone, err, B_TRUE);
6369                return;
6370        }
6371        while (zonecfg_getadminent(handle, &admintab) == Z_OK) {
6372                check_reqd_prop(admintab.zone_admin_user, RT_ADMIN,
6373                    PT_USER, &ret_val);
6374                check_reqd_prop(admintab.zone_admin_auths, RT_ADMIN,
6375                    PT_AUTHS, &ret_val);
6376                if ((ret_val == Z_OK) && (getpwnam(admintab.zone_admin_user)
6377                    == NULL)) {
6378                        zerr(gettext("%s %s is not a valid username"),
6379                            pt_to_str(PT_USER),
6380                            admintab.zone_admin_user);
6381                        ret_val = Z_BAD_PROPERTY;
6382                }
6383                if ((ret_val == Z_OK) && (!zonecfg_valid_auths(
6384                    admintab.zone_admin_auths, zone))) {
6385                        ret_val = Z_BAD_PROPERTY;
6386                }
6387        }
6388        (void) zonecfg_endadminent(handle);

6390        if (zonecfg_getsecflagsent(handle, &secflagstab) == Z_OK) {
2900        if ((err = zonecfg_getsecflagsent(handle, &secflagstab)) != Z_OK) {
2901                zone_perror(zone, err, B_TRUE);
2902                return;
2903        }

6391                /*
6392                 * No properties are required, but any specified should be
6393                 * valid
6394                 */
6395                if (verify_secflags(&secflagstab) != B_TRUE) {
6396                        /* Error is reported from verify_secflags */
6397                        ret_val = Z_BAD_PROPERTY;
6398                }
6399        }
6400 #endif /* ! codereview */

6402        if (!global_scope) {
6403                zerr(gettext("resource specification incomplete"));
6404                saw_error = B_TRUE;
6405                if (ret_val == Z_OK)
6406                        ret_val = Z_INSUFFICIENT_SPEC;
6407        }

6409        if (save) {
6410                if (ret_val == Z_OK) {
```



```
6411                        if ((ret_val = zonecfg_save(handle)) == Z_OK) {
6412                                need_to_commit = B_FALSE;
6413                                (void) strlcpy(revert_zone, zone,
6414                                    sizeof (revert_zone));
6415                        }
6416                } else {
6417                        zerr(gettext("Zone %s failed to verify"), zone);
6418                }
6419        }
6420        if (ret_val != Z_OK)
6421                zone_perror(zone, ret_val, B_TRUE);
6422 }

6424 void
6425 cancel_func(cmd_t *cmd)
6426 {
6427        int arg;
6428        boolean_t arg_err = B_FALSE;

6430        assert(cmd != NULL);

6432        optind = 0;
6433        while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "?")) != EOF) {
6434                switch (arg) {
6435                case '?':
6436                        longer_usage(CMD_CANCEL);
6437                        arg_err = B_TRUE;
6438                        break;
6439                default:
6440                        short_usage(CMD_CANCEL);
6441                        arg_err = B_TRUE;
6442                        break;
6443                }
6444        }
6445        if (arg_err)
6446                return;

6448        if (optind != cmd->cmd_argc) {
6449                short_usage(CMD_CANCEL);
6450                return;
6451        }

6453        if (global_scope)
6454                scope_usage(CMD_CANCEL);
6455        global_scope = B_TRUE;
6456        zonecfg_free_fs_option_list(in_progress_fstab.zone_fs_options);
6457        bzero(&in_progress_fstab, sizeof (in_progress_fstab));
6458        bzero(&in_progress_nwiftab, sizeof (in_progress_nwiftab));
6459        bzero(&in_progress_devtab, sizeof (in_progress_devtab));
6460        zonecfg_free_rctl_value_list(in_progress_rctltab.zone_rctl_valptr);
6461        bzero(&in_progress_rctltab, sizeof (in_progress_rctltab));
6462        bzero(&in_progress_attrtab, sizeof (in_progress_attrtab));
6463        bzero(&in_progress_dstab, sizeof (in_progress_dstab));
6464 }

6466 static int
6467 validate_attr_name(char *name)
6468 {
6469        int i;

6471        if (!isalnum(name[0])) {
6472                zerr(gettext("Invalid %s %s %s: must start with an alpha-"
6473                    "numeric character."), rt_to_str(RT_ATTR),
6474                    pt_to_str(PT_NAME), name);
6475                return (Z_INVAL);
6476        }
```

```
6477         for (i = 1; name[i]; i++)
6478                 if (!isalnum(name[i]) && name[i] != '-' && name[i] != '.') {
6479                         zerr(gettext("Invalid %s %s %s: can only contain "
6480                             "alpha-numeric characters, plus '-' and '.'."),
6481                             rt_to_str(RT_ATTR), pt_to_str(PT_NAME), name);
6482                         return (Z_INVAL);
6483                 }
6484         return (Z_OK);
6485 }

6487 static int
6488 validate_attr_type_val(struct zone_attrtab *attrtab)
6489 {
6490         boolean_t boolval;
6491         int64_t intval;
6492         char strval[MAXNAMELEN];
6493         uint64_t uintval;

6495         if (strcmp(attrtab->zone_attr_type, "boolean") == 0) {
6496                 if (zonecfg_get_attr_boolean(attrtab, &boolval) == Z_OK)
6497                         return (Z_OK);
6498                 zerr(gettext("invalid %s value for %s=%s"),
6499                     rt_to_str(RT_ATTR), pt_to_str(PT_TYPE), "boolean");
6500                 return (Z_ERR);
6501         }

6503         if (strcmp(attrtab->zone_attr_type, "int") == 0) {
6504                 if (zonecfg_get_attr_int(attrtab, &intval) == Z_OK)
6505                         return (Z_OK);
6506                 zerr(gettext("invalid %s value for %s=%s"),
6507                     rt_to_str(RT_ATTR), pt_to_str(PT_TYPE), "int");
6508                 return (Z_ERR);
6509         }

6511         if (strcmp(attrtab->zone_attr_type, "string") == 0) {
6512                 if (zonecfg_get_attr_string(attrtab, strval,
6513                     sizeof (strval)) == Z_OK)
6514                         return (Z_OK);
6515                 zerr(gettext("invalid %s value for %s=%s"),
6516                     rt_to_str(RT_ATTR), pt_to_str(PT_TYPE), "string");
6517                 return (Z_ERR);
6518         }

6520         if (strcmp(attrtab->zone_attr_type, "uint") == 0) {
6521                 if (zonecfg_get_attr_uint(attrtab, &uintval) == Z_OK)
6522                         return (Z_OK);
6523                 zerr(gettext("invalid %s value for %s=%s"),
6524                     rt_to_str(RT_ATTR), pt_to_str(PT_TYPE), "uint");
6525                 return (Z_ERR);
6526         }

6528         zerr(gettext("invalid %s %s '%s'"), rt_to_str(RT_ATTR),
6529             pt_to_str(PT_TYPE), attrtab->zone_attr_type);
6530         return (Z_ERR);
6531 }

6533 /*
6534  * Helper function for end_func-- checks the existence of a given property
6535  * and emits a message if not specified.
6536  */
6537 static int
6538 end_check_reqd(char *attr, int pt, boolean_t *validation_failed)
6539 {
6540         if (strlen(attr) == 0) {
6541                 *validation_failed = B_TRUE;
6542                 zerr(gettext("%s not specified"), pt_to_str(pt));
```



```
6543                 return (Z_ERR);
6544         }
6545         return (Z_OK);
6546 }

6548 static void
6549 net_exists_error(struct zone_nwiftab nwif)
6550 {
6551         if (strlen(nwif.zone_nwif_address) > 0) {
6552                 zerr(gettext("A %s resource with the %s '%s', "
6553                     "and %s '%s' already exists."),
6554                     rt_to_str(RT_NET),
6555                     pt_to_str(PT_PHYSICAL),
6556                     nwif.zone_nwif_physical,
6557                     pt_to_str(PT_ADDRESS),
6558                     in_progress_nwiftab.zone_nwif_address);
6559         } else {
6560                 zerr(gettext("A %s resource with the %s '%s', "
6561                     "and %s '%s' already exists."),
6562                     rt_to_str(RT_NET),
6563                     pt_to_str(PT_PHYSICAL),
6564                     nwif.zone_nwif_physical,
6565                     pt_to_str(PT_ALLOWED_ADDRESS),
6566                     nwif.zone_nwif_allowed_address);
6567         }
6568 }

6570 void
6571 end_func(cmd_t *cmd)
6572 {
6573         boolean_t validation_failed = B_FALSE;
6574         boolean_t arg_err = B_FALSE;
6575         struct zone_fstab tmp_fstab;
6576         struct zone_nwiftab tmp_nwiftab;
6577         struct zone_devtab tmp_devtab;
6578         struct zone_rctltab tmp_rctltab;
6579         struct zone_attrtab tmp_attrtab;
6580         struct zone_dstab tmp_dstab;
6581         struct zone_admintab tmp_admintab;
6582         int err, arg, res1, res2, res3;
6583         uint64_t swap_limit;
6584         uint64_t locked_limit;
6585         uint64_t proc_cap;

6587         assert(cmd != NULL);

6589         optind = 0;
6590         while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "?")) != EOF) {
6591                 switch (arg) {
6592                 case '?':
6593                         longer_usage(CMD_END);
6594                         arg_err = B_TRUE;
6595                         break;
6596                 default:
6597                         short_usage(CMD_END);
6598                         arg_err = B_TRUE;
6599                         break;
6600                 }
6601         }
6602         if (arg_err)
6603                 return;

6605         if (optind != cmd->cmd_argc) {
6606                 short_usage(CMD_END);
6607                 return;
6608         }
```

```
6610         if (global_scope) {
6611                 scope_usage(CMD_END);
6612                 return;
6613         }

6615         assert(end_op == CMD_ADD || end_op == CMD_SELECT);

6617         switch (resource_scope) {
6618         case RT_FS:
6619                 /* First make sure everything was filled in. */
6620                 if (end_check_reqd(in_progress_fstab.zone_fs_dir,
6621                     PT_DIR, &validation_failed) == Z_OK) {
6622                         if (in_progress_fstab.zone_fs_dir[0] != '/') {
6623                                 zerr(gettext("%s %s is not an absolute path."),
6624                                     pt_to_str(PT_DIR),
6625                                     in_progress_fstab.zone_fs_dir);
6626                                 validation_failed = B_TRUE;
6627                         }
6628                 }

6630                 (void) end_check_reqd(in_progress_fstab.zone_fs_special,
6631                     PT_SPECIAL, &validation_failed);

6633                 if (in_progress_fstab.zone_fs_raw[0] != '\0' &&
6634                     in_progress_fstab.zone_fs_raw[0] != '/') {
6635                         zerr(gettext("%s %s is not an absolute path."),
6636                             pt_to_str(PT_RAW),
6637                             in_progress_fstab.zone_fs_raw);
6638                         validation_failed = B_TRUE;
6639                 }

6641                 (void) end_check_reqd(in_progress_fstab.zone_fs_type, PT_TYPE,
6642                     &validation_failed);

6644                 if (validation_failed) {
6645                         saw_error = B_TRUE;
6646                         return;
6647                 }

6649                 if (end_op == CMD_ADD) {
6650                         /* Make sure there isn't already one like this. */
6651                         bzero(&tmp_fstab, sizeof (tmp_fstab));
6652                         (void) strlcpy(tmp_fstab.zone_fs_dir,
6653                             in_progress_fstab.zone_fs_dir,
6654                             sizeof (tmp_fstab.zone_fs_dir));
6655                         err = zonecfg_lookup_filesystem(handle, &tmp_fstab);
6656                         zonecfg_free_fs_option_list(tmp_fstab.zone_fs_options);
6657                         if (err == Z_OK) {
6658                                 zerr(gettext("A %s resource "
6659                                     "with the %s '%s' already exists."),
6660                                     rt_to_str(RT_FS), pt_to_str(PT_DIR),
6661                                     in_progress_fstab.zone_fs_dir);
6662                                 saw_error = B_TRUE;
6663                                 return;
6664                         }
6665                         err = zonecfg_add_filesystem(handle,
6666                             &in_progress_fstab);
6667                 } else {
6668                         err = zonecfg_modify_filesystem(handle, &old_fstab,
6669                             &in_progress_fstab);
6670                 }
6671                 zonecfg_free_fs_option_list(in_progress_fstab.zone_fs_options);
6672                 in_progress_fstab.zone_fs_options = NULL;
6673                 break;
```



```
6675         case RT_NET:
6676                 /*
6677                  * First make sure everything was filled in.
6678                  * Since we don't know whether IP will be shared
6679                  * or exclusive here, some checks are deferred until
6680                  * the verify command.
6681                  */
6682                 (void) end_check_reqd(in_progress_nwiftab.zone_nwif_physical,
6683                     PT_PHYSICAL, &validation_failed);

6685                 if (validation_failed) {
6686                         saw_error = B_TRUE;
6687                         return;
6688                 }
6689                 if (end_op == CMD_ADD) {
6690                         /* Make sure there isn't already one like this. */
6691                         bzero(&tmp_nwiftab, sizeof (tmp_nwiftab));
6692                         (void) strlcpy(tmp_nwiftab.zone_nwif_physical,
6693                             in_progress_nwiftab.zone_nwif_physical,
6694                             sizeof (tmp_nwiftab.zone_nwif_physical));
6695                         (void) strlcpy(tmp_nwiftab.zone_nwif_address,
6696                             in_progress_nwiftab.zone_nwif_address,
6697                             sizeof (tmp_nwiftab.zone_nwif_address));
6698                         (void) strlcpy(tmp_nwiftab.zone_nwif_allowed_address,
6699                             in_progress_nwiftab.zone_nwif_allowed_address,
6700                             sizeof (tmp_nwiftab.zone_nwif_allowed_address));
6701                         (void) strlcpy(tmp_nwiftab.zone_nwif_defrouter,
6702                             in_progress_nwiftab.zone_nwif_defrouter,
6703                             sizeof (tmp_nwiftab.zone_nwif_defrouter));
6704                         if (zonecfg_lookup_nwif(handle, &tmp_nwiftab) == Z_OK) {
6705                                 net_exists_error(in_progress_nwiftab);
6706                                 saw_error = B_TRUE;
6707                                 return;
6708                         }
6709                         err = zonecfg_add_nwif(handle, &in_progress_nwiftab);
6710                 } else {
6711                         err = zonecfg_modify_nwif(handle, &old_nwiftab,
6712                             &in_progress_nwiftab);
6713                 }
6714                 break;

6716         case RT_DEVICE:
6717                 /* First make sure everything was filled in. */
6718                 (void) end_check_reqd(in_progress_devtab.zone_dev_match,
6719                     PT_MATCH, &validation_failed);

6721                 if (validation_failed) {
6722                         saw_error = B_TRUE;
6723                         return;
6724                 }

6726                 if (end_op == CMD_ADD) {
6727                         /* Make sure there isn't already one like this. */
6728                         (void) strlcpy(tmp_devtab.zone_dev_match,
6729                             in_progress_devtab.zone_dev_match,
6730                             sizeof (tmp_devtab.zone_dev_match));
6731                         if (zonecfg_lookup_dev(handle, &tmp_devtab) == Z_OK) {
6732                                 zerr(gettext("A %s resource with the %s '%s' "
6733                                     "already exists."), rt_to_str(RT_DEVICE),
6734                                     pt_to_str(PT_MATCH),
6735                                     in_progress_devtab.zone_dev_match);
6736                                 saw_error = B_TRUE;
6737                                 return;
6738                         }
6739                         err = zonecfg_add_dev(handle, &in_progress_devtab);
6740                 } else {
```

```
6741                        err = zonecfg_modify_dev(handle, &old_devtab,
6742                            &in_progress_devtab);
6743                }
6744                break;

6746        case RT_RCTL:
6747                /* First make sure everything was filled in. */
6748                (void) end_check_reqd(in_progress_rctltab.zone_rctl_name,
6749                    PT_NAME, &validation_failed);

6751                if (in_progress_rctltab.zone_rctl_valptr == NULL) {
6752                        zerr(gettext("no %s specified"), pt_to_str(PT_VALUE));
6753                        validation_failed = B_TRUE;
6754                }

6756                if (validation_failed) {
6757                        saw_error = B_TRUE;
6758                        return;
6759                }

6761                if (end_op == CMD_ADD) {
6762                        /* Make sure there isn't already one like this. */
6763                        (void) strlcpy(tmp_rctltab.zone_rctl_name,
6764                            in_progress_rctltab.zone_rctl_name,
6765                            sizeof (tmp_rctltab.zone_rctl_name));
6766                        tmp_rctltab.zone_rctl_valptr = NULL;
6767                        err = zonecfg_lookup_rctl(handle, &tmp_rctltab);
6768                        zonecfg_free_rctl_value_list(
6769                            tmp_rctltab.zone_rctl_valptr);
6770                        if (err == Z_OK) {
6771                                zerr(gettext("A %s resource "
6772                                    "with the %s '%s' already exists."),
6773                                    rt_to_str(RT_RCTL), pt_to_str(PT_NAME),
6774                                    in_progress_rctltab.zone_rctl_name);
6775                                saw_error = B_TRUE;
6776                                return;
6777                        }
6778                        err = zonecfg_add_rctl(handle, &in_progress_rctltab);
6779                } else {
6780                        err = zonecfg_modify_rctl(handle, &old_rctltab,
6781                            &in_progress_rctltab);
6782                }
6783                if (err == Z_OK) {
6784                        zonecfg_free_rctl_value_list(
6785                            in_progress_rctltab.zone_rctl_valptr);
6786                        in_progress_rctltab.zone_rctl_valptr = NULL;
6787                }
6788                break;

6790        case RT_ATTR:
6791                /* First make sure everything was filled in. */
6792                (void) end_check_reqd(in_progress_attrtab.zone_attr_name,
6793                    PT_NAME, &validation_failed);
6794                (void) end_check_reqd(in_progress_attrtab.zone_attr_type,
6795                    PT_TYPE, &validation_failed);
6796                (void) end_check_reqd(in_progress_attrtab.zone_attr_value,
6797                    PT_VALUE, &validation_failed);

6799                if (validate_attr_name(in_progress_attrtab.zone_attr_name) !=
6800                    Z_OK)
6801                        validation_failed = B_TRUE;

6803                if (validate_attr_type_val(&in_progress_attrtab) != Z_OK)
6804                        validation_failed = B_TRUE;

6806                if (validation_failed) {
```



```
6807                        saw_error = B_TRUE;
6808                        return;
6809                }
6810                if (end_op == CMD_ADD) {
6811                        /* Make sure there isn't already one like this. */
6812                        bzero(&tmp_attrtab, sizeof (tmp_attrtab));
6813                        (void) strlcpy(tmp_attrtab.zone_attr_name,
6814                            in_progress_attrtab.zone_attr_name,
6815                            sizeof (tmp_attrtab.zone_attr_name));
6816                        if (zonecfg_lookup_attr(handle, &tmp_attrtab) == Z_OK) {
6817                                zerr(gettext("An %s resource "
6818                                    "with the %s '%s' already exists."),
6819                                    rt_to_str(RT_ATTR), pt_to_str(PT_NAME),
6820                                    in_progress_attrtab.zone_attr_name);
6821                                saw_error = B_TRUE;
6822                                return;
6823                        }
6824                        err = zonecfg_add_attr(handle, &in_progress_attrtab);
6825                } else {
6826                        err = zonecfg_modify_attr(handle, &old_attrtab,
6827                            &in_progress_attrtab);
6828                }
6829                break;
6830        case RT_DATASET:
6831                /* First make sure everything was filled in. */
6832                if (strlen(in_progress_dstab.zone_dataset_name) == 0) {
6833                        zerr("%s %s", pt_to_str(PT_NAME),
6834                            gettext("not specified"));
6835                        saw_error = B_TRUE;
6836                        validation_failed = B_TRUE;
6837                }
6838                if (validation_failed)
6839                        return;
6840                if (end_op == CMD_ADD) {
6841                        /* Make sure there isn't already one like this. */
6842                        bzero(&tmp_dstab, sizeof (tmp_dstab));
6843                        (void) strlcpy(tmp_dstab.zone_dataset_name,
6844                            in_progress_dstab.zone_dataset_name,
6845                            sizeof (tmp_dstab.zone_dataset_name));
6846                        err = zonecfg_lookup_ds(handle, &tmp_dstab);
6847                        if (err == Z_OK) {
6848                                zerr(gettext("A %s resource "
6849                                    "with the %s '%s' already exists."),
6850                                    rt_to_str(RT_DATASET), pt_to_str(PT_NAME),
6851                                    in_progress_dstab.zone_dataset_name);
6852                                saw_error = B_TRUE;
6853                                return;
6854                        }
6855                        err = zonecfg_add_ds(handle, &in_progress_dstab);
6856                } else {
6857                        err = zonecfg_modify_ds(handle, &old_dstab,
6858                            &in_progress_dstab);
6859                }
6860                break;
6861        case RT_DCPU:
6862                /* Make sure everything was filled in. */
6863                if (end_check_reqd(in_progress_psettab.zone_ncpu_min,
6864                    PT_NCPUS, &validation_failed) != Z_OK) {
6865                        saw_error = B_TRUE;
6866                        return;
6867                }

6869                if (end_op == CMD_ADD) {
6870                        err = zonecfg_add_pset(handle, &in_progress_psettab);
6871                } else {
6872                        err = zonecfg_modify_pset(handle, &in_progress_psettab);
```

```
6873                }
6874                break;
6875        case RT_PCAP:
6876                /* Make sure everything was filled in. */
6877                if (zonecfg_get_aliased_rctl(handle, ALIAS_CPUCAP, &proc_cap)
6878                    != Z_OK) {
6879                        zerr(gettext("%s not specified"), pt_to_str(PT_NCPUS));
6880                        saw_error = B_TRUE;
6881                        validation_failed = B_TRUE;
6882                        return;
6883                }
6884                err = Z_OK;
6885                break;
6886        case RT_MCAP:
6887                /* Make sure everything was filled in. */
6888                res1 = strlen(in_progress_mcaptab.zone_physmem_cap) == 0 ?
6889                    Z_ERR : Z_OK;
6890                res2 = zonecfg_get_aliased_rctl(handle, ALIAS_MAXSWAP,
6891                    &swap_limit);
6892                res3 = zonecfg_get_aliased_rctl(handle, ALIAS_MAXLOCKEDMEM,
6893                    &locked_limit);

6895                if (res1 != Z_OK && res2 != Z_OK && res3 != Z_OK) {
6896                        zerr(gettext("No property was specified.  One of %s, "
6897                            "%s or %s is required."), pt_to_str(PT_PHYSICAL),
6898                            pt_to_str(PT_SWAP), pt_to_str(PT_LOCKED));
6899                        saw_error = B_TRUE;
6900                        return;
6901                }

6903                /* if phys & locked are both set, verify locked <= phys */
6904                if (res1 == Z_OK && res3 == Z_OK) {
6905                        uint64_t phys_limit;
6906                        char *endp;

6908                        phys_limit = strtoull(
6909                            in_progress_mcaptab.zone_physmem_cap, &endp, 10);
6910                        if (phys_limit < locked_limit) {
6911                                zerr(gettext("The %s cap must be less than or "
6912                                    "equal to the %s cap."),
6913                                    pt_to_str(PT_LOCKED),
6914                                    pt_to_str(PT_PHYSICAL));
6915                                saw_error = B_TRUE;
6916                                return;
6917                        }
6918                }

6920                err = Z_OK;
6921                if (res1 == Z_OK) {
6922                        /*
6923                         * We could be ending from either an add operation
6924                         * or a select operation.  Since all of the properties
6925                         * within this resource are optional, we always use
6926                         * modify on the mcap entry.  zonecfg_modify_mcap()
6927                         * will handle both adding and modifying a memory cap.
6928                         */
6929                        err = zonecfg_modify_mcap(handle, &in_progress_mcaptab);
6930                } else if (end_op == CMD_SELECT) {
6931                        /*
6932                         * If we're ending from a select and the physical
6933                         * memory cap is empty then the user could have cleared
6934                         * the physical cap value, so try to delete the entry.
6935                         */
6936                        (void) zonecfg_delete_mcap(handle);
6937                }
6938                break;
```



```
6939        case RT_ADMIN:
6940                /* First make sure everything was filled in. */
6941                if (end_check_reqd(in_progress_admintab.zone_admin_user,
6942                    PT_USER, &validation_failed) == Z_OK) {
6943                        if (getpwnam(in_progress_admintab.zone_admin_user)
6944                            == NULL) {
6945                                zerr(gettext("%s %s is not a valid username"),
6946                                    pt_to_str(PT_USER),
6947                                    in_progress_admintab.zone_admin_user);
6948                                validation_failed = B_TRUE;
6949                        }
6950                }

6952                if (end_check_reqd(in_progress_admintab.zone_admin_auths,
6953                    PT_AUTHS, &validation_failed) == Z_OK) {
6954                        if (!zonecfg_valid_auths(
6955                            in_progress_admintab.zone_admin_auths,
6956                            zone)) {
6957                                validation_failed = B_TRUE;
6958                        }
6959                }

6961                if (validation_failed) {
6962                        saw_error = B_TRUE;
6963                        return;
6964                }

6966                if (end_op == CMD_ADD) {
6967                        /* Make sure there isn't already one like this. */
6968                        bzero(&tmp_admintab, sizeof (tmp_admintab));
6969                        (void) strlcpy(tmp_admintab.zone_admin_user,
6970                            in_progress_admintab.zone_admin_user,
6971                            sizeof (tmp_admintab.zone_admin_user));
6972                        err = zonecfg_lookup_admin(
6973                            handle, &tmp_admintab);
6974                        if (err == Z_OK) {
6975                                zerr(gettext("A %s resource "
6976                                    "with the %s '%s' already exists."),
6977                                    rt_to_str(RT_ADMIN),
6978                                    pt_to_str(PT_USER),
6979                                    in_progress_admintab.zone_admin_user);
6980                                saw_error = B_TRUE;
6981                                return;
6982                        }
6983                        err = zonecfg_add_admin(handle,
6984                            &in_progress_admintab, zone);
6985                } else {
6986                        err = zonecfg_modify_admin(handle,
6987                            &old_admintab, &in_progress_admintab,
6988                            zone);
6989                }
6990                break;
6991        case RT_SECFLAGS:
6992                if (verify_secflags(&in_progress_secflagstab) != B_TRUE) {
6993                        saw_error = B_TRUE;
6994                        return;
6995                }

6997                if (end_op == CMD_ADD) {
6998                        err = zonecfg_add_secflags(handle,
6999                            &in_progress_secflagstab);
7000                } else {
7001                        err = zonecfg_modify_secflags(handle,
7002                            &old_secflagstab, &in_progress_secflagstab);
7003                }
7004                break;
```

```
7005         default:
7006                 zone_perror(rt_to_str(resource_scope), Z_NO_RESOURCE_TYPE,
7007                     B_TRUE);
7008                 saw_error = B_TRUE;
7009                 return;
7010         }

7012         if (err != Z_OK) {
7013                 zone_perror(zone, err, B_TRUE);
7014         } else {
7015                 need_to_commit = B_TRUE;
7016                 global_scope = B_TRUE;
7017                 end_op = -1;
7018         }
7019 }

7021 void
7022 commit_func(cmd_t *cmd)
7023 {
7024         int arg;
7025         boolean_t arg_err = B_FALSE;

7027         optind = 0;
7028         while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "?")) != EOF) {
7029                 switch (arg) {
7030                 case '?':
7031                         longer_usage(CMD_COMMIT);
7032                         arg_err = B_TRUE;
7033                         break;
7034                 default:
7035                         short_usage(CMD_COMMIT);
7036                         arg_err = B_TRUE;
7037                         break;
7038                 }
7039         }
7040         if (arg_err)
7041                 return;

7043         if (optind != cmd->cmd_argc) {
7044                 short_usage(CMD_COMMIT);
7045                 return;
7046         }

7048         if (zone_is_read_only(CMD_COMMIT))
7049                 return;

7051         assert(cmd != NULL);

7053         cmd->cmd_argc = 1;
7054         /*
7055          * cmd_arg normally comes from a strdup() in the lexer, and the
7056          * whole cmd structure and its (char *) attributes are freed at
7057          * the completion of each command, so the strdup() below is needed
7058          * to match this and prevent a core dump from trying to free()
7059          * something that can't be.
7060          */
7061         if ((cmd->cmd_argv[0] = strdup("save")) == NULL) {
7062                 zone_perror(zone, Z_NOMEM, B_TRUE);
7063                 exit(Z_ERR);
7064         }
7065         cmd->cmd_argv[1] = NULL;
7066         verify_func(cmd);
7067 }

7069 void
7070 revert_func(cmd_t *cmd)
```



```
7071 {
7072         char line[128]; /* enough to ask a question */
7073         boolean_t force = B_FALSE;
7074         boolean_t arg_err = B_FALSE;
7075         int err, arg, answer;

7077         optind = 0;
7078         while ((arg = getopt(cmd->cmd_argc, cmd->cmd_argv, "?F")) != EOF) {
7079                 switch (arg) {
7080                 case '?':
7081                         longer_usage(CMD_REVERT);
7082                         arg_err = B_TRUE;
7083                         break;
7084                 case 'F':
7085                         force = B_TRUE;
7086                         break;
7087                 default:
7088                         short_usage(CMD_REVERT);
7089                         arg_err = B_TRUE;
7090                         break;
7091                 }
7092         }
7093         if (arg_err)
7094                 return;

7096         if (optind != cmd->cmd_argc) {
7097                 short_usage(CMD_REVERT);
7098                 return;
7099         }

7101         if (zone_is_read_only(CMD_REVERT))
7102                 return;

7104         if (!global_scope) {
7105                 zerr(gettext("You can only use %s in the global scope.\nUse"
7106                     " '%s' to cancel changes to a resource specification."),
7107                     cmd_to_str(CMD_REVERT), cmd_to_str(CMD_CANCEL));
7108                 saw_error = B_TRUE;
7109                 return;
7110         }

7112         if (zonecfg_check_handle(handle) != Z_OK) {
7113                 zerr(gettext("No changes to revert."));
7114                 saw_error = B_TRUE;
7115                 return;
7116         }

7118         if (!force) {
7119                 (void) snprintf(line, sizeof (line),
7120                     gettext("Are you sure you want to revert"));
7121                 if ((answer = ask_yesno(B_FALSE, line)) == -1) {
7122                         zerr(gettext("Input not from terminal and -F not "
7123                             "specified:\n%s command ignored, exiting."),
7124                             cmd_to_str(CMD_REVERT));
7125                         exit(Z_ERR);
7126                 }
7127                 if (answer != 1)
7128                         return;
7129         }

7131         /*
7132          * Reset any pending admins that were
7133          * removed from the previous zone
7134          */
7135         zonecfg_remove_userauths(handle, "", zone, B_FALSE);
```

```
7137         /*
7138          * Time for a new handle: finish the old one off first
7139          * then get a new one properly to avoid leaks.
7140          */
7141         zonecfg_fini_handle(handle);
7142         if ((handle = zonecfg_init_handle()) == NULL) {
7143                 zone_perror(execname, Z_NOMEM, B_TRUE);
7144                 exit(Z_ERR);
7145         }

7147         if ((err = zonecfg_get_handle(revert_zone, handle)) != Z_OK) {
7148                 saw_error = B_TRUE;
7149                 got_handle = B_FALSE;
7150                 if (err == Z_NO_ZONE)
7151                         zerr(gettext("%s: no such saved zone to revert to."),
7152                             revert_zone);
7153                 else
7154                         zone_perror(zone, err, B_TRUE);
7155         }
7156         (void) strlcpy(zone, revert_zone, sizeof (zone));
7157 }

7159 void
7160 help_func(cmd_t *cmd)
7161 {
7162         int i;

7164         assert(cmd != NULL);

7166         if (cmd->cmd_argc == 0) {
7167                 usage(B_TRUE, global_scope ? HELP_SUBCMDS : HELP_RES_SCOPE);
7168                 return;
7169         }
7170         if (strcmp(cmd->cmd_argv[0], "usage") == 0) {
7171                 usage(B_TRUE, HELP_USAGE);
7172                 return;
7173         }
7174         if (strcmp(cmd->cmd_argv[0], "commands") == 0) {
7175                 usage(B_TRUE, HELP_SUBCMDS);
7176                 return;
7177         }
7178         if (strcmp(cmd->cmd_argv[0], "syntax") == 0) {
7179                 usage(B_TRUE, HELP_SYNTAX | HELP_RES_PROPS);
7180                 return;
7181         }
7182         if (strcmp(cmd->cmd_argv[0], "-?") == 0) {
7183                 longer_usage(CMD_HELP);
7184                 return;
7185         }

7187         for (i = 0; i <= CMD_MAX; i++) {
7188                 if (strcmp(cmd->cmd_argv[0], cmd_to_str(i)) == 0) {
7189                         longer_usage(i);
7190                         return;
7191                 }
7192         }
7193         /* We do not use zerr() here because we do not want its extra \n. */
7194         (void) fprintf(stderr, gettext("Unknown help subject %s.  "),
7195             cmd->cmd_argv[0]);
7196         usage(B_FALSE, HELP_META);
7197 }

7199 static int
7200 string_to_yyin(char *string)
7201 {
7202         if ((yyin = tmpfile()) == NULL) {
```



```
7203                 zone_perror(execname, Z_TEMP_FILE, B_TRUE);
7204                 return (Z_ERR);
7205         }
7206         if (fwrite(string, strlen(string), 1, yyin) != 1) {
7207                 zone_perror(execname, Z_TEMP_FILE, B_TRUE);
7208                 return (Z_ERR);
7209         }
7210         if (fseek(yyin, 0, SEEK_SET) != 0) {
7211                 zone_perror(execname, Z_TEMP_FILE, B_TRUE);
7212                 return (Z_ERR);
7213         }
7214         return (Z_OK);
7215 }

7217 /* This is the back-end helper function for read_input() below. */

7219 static int
7220 cleanup()
7221 {
7222         int answer;
7223         cmd_t *cmd;

7225         if (!interactive_mode && !cmd_file_mode) {
7226                 /*
7227                  * If we're not in interactive mode, and we're not in command
7228                  * file mode, then we must be in commands-from-the-command-line
7229                  * mode.  As such, we can't loop back and ask for more input.
7230                  * It was OK to prompt for such things as whether or not to
7231                  * really delete a zone in the command handler called from
7232                  * yyparse() above, but "really quit?" makes no sense in this
7233                  * context.  So disable prompting.
7234                  */
7235                 ok_to_prompt = B_FALSE;
7236         }
7237         if (!global_scope) {
7238                 if (!time_to_exit) {
7239                         /*
7240                          * Just print a simple error message in the -1 case,
7241                          * since exit_func() already handles that case, and
7242                          * EOF means we are finished anyway.
7243                          */
7244                         answer = ask_yesno(B_FALSE,
7245                             gettext("Resource incomplete; really quit"));
7246                         if (answer == -1) {
7247                                 zerr(gettext("Resource incomplete."));
7248                                 return (Z_ERR);
7249                         }
7250                         if (answer != 1) {
7251                                 yyin = stdin;
7252                                 return (Z_REPEAT);
7253                         }
7254                 } else {
7255                         saw_error = B_TRUE;
7256                 }
7257         }
7258         /*
7259          * Make sure we tried something and that the handle checks
7260          * out, or we would get a false error trying to commit.
7261          */
7262         if (need_to_commit && zonecfg_check_handle(handle) == Z_OK) {
7263                 if ((cmd = alloc_cmd()) == NULL) {
7264                         zone_perror(zone, Z_NOMEM, B_TRUE);
7265                         return (Z_ERR);
7266                 }
7267                 cmd->cmd_argc = 0;
7268                 cmd->cmd_argv[0] = NULL;
```

```
7269                commit_func(cmd);
7270                free_cmd(cmd);
7271                /*
7272                 * need_to_commit will get set back to FALSE if the
7273                 * configuration is saved successfully.
7274                 */
7275                if (need_to_commit) {
7276                        if (force_exit) {
7277                                zerr(gettext("Configuration not saved."));
7278                                return (Z_ERR);
7279                        }
7280                        answer = ask_yesno(B_FALSE,
7281                            gettext("Configuration not saved; really quit"));
7282                        if (answer == -1) {
7283                                zerr(gettext("Configuration not saved."));
7284                                return (Z_ERR);
7285                        }
7286                        if (answer != 1) {
7287                                time_to_exit = B_FALSE;
7288                                yyin = stdin;
7289                                return (Z_REPEAT);
7290                        }
7291                }
7292        }
7293        return ((need_to_commit || saw_error) ? Z_ERR : Z_OK);
7294 }

7296 /*
7297  * read_input() is the driver of this program.  It is a wrapper around
7298  * yyparse(), printing appropriate prompts when needed, checking for
7299  * exit conditions and reacting appropriately [the latter in its cleanup()
7300  * helper function].
7301  *
7302  * Like most zonecfg functions, it returns Z_OK or Z_ERR, *or* Z_REPEAT
7303  * so do_interactive() knows that we are not really done (i.e, we asked
7304  * the user if we should really quit and the user said no).
7305  */
7306 static int
7307 read_input()
7308 {
7309        boolean_t yyin_is_a_tty = isatty(fileno(yyin));
7310        /*
7311         * The prompt is "e:z> " or "e:z:r> " where e is execname, z is zone
7312         * and r is resource_scope: 5 is for the two ":"s + "> " + terminator.
7313         */
7314        char prompt[MAXPATHLEN + ZONENAME_MAX + MAX_RT_STRLEN + 5], *line;

7316        /* yyin should have been set to the appropriate (FILE *) if not stdin */
7317        newline_terminated = B_TRUE;
7318        for (;;) {
7319                if (yyin_is_a_tty) {
7320                        if (newline_terminated) {
7321                                if (global_scope)
7322                                        (void) snprintf(prompt, sizeof (prompt),
7323                                            "%s:%s> ", execname, zone);
7324                                else
7325                                        (void) snprintf(prompt, sizeof (prompt),
7326                                            "%s:%s:%s> ", execname, zone,
7327                                            rt_to_str(resource_scope));
7328                        }
7329                        /*
7330                         * If the user hits ^C then we want to catch it and
7331                         * start over.  If the user hits EOF then we want to
7332                         * bail out.
7333                         */
7334                        line = gl_get_line(gl, prompt, NULL, -1);
```



```
7335                        if (gl_return_status(gl) == GLR_SIGNAL) {
7336                                gl_abandon_line(gl);
7337                                continue;
7338                        }
7339                        if (line == NULL)
7340                                break;
7341                        (void) string_to_yyin(line);
7342                        while (!feof(yyin))
7343                                yyparse();
7344                } else {
7345                        yyparse();
7346                }
7347                /* Bail out on an error in command file mode. */
7348                if (saw_error && cmd_file_mode && !interactive_mode)
7349                        time_to_exit = B_TRUE;
7350                if (time_to_exit || (!yyin_is_a_tty && feof(yyin)))
7351                        break;
7352        }
7353        return (cleanup());
7354 }

7356 /*
7357  * This function is used in the zonecfg-interactive-mode scenario: it just
7358  * calls read_input() until we are done.
7359  */

7361 static int
7362 do_interactive(void)
7363 {
7364        int err;

7366        interactive_mode = B_TRUE;
7367        if (!read_only_mode) {
7368                /*
7369                 * Try to set things up proactively in interactive mode, so
7370                 * that if the zone in question does not exist yet, we can
7371                 * provide the user with a clue.
7372                 */
7373                (void) initialize(B_FALSE);
7374        }
7375        do {
7376                err = read_input();
7377        } while (err == Z_REPEAT);
7378        return (err);
7379 }

7381 /*
7382  * cmd_file is slightly more complicated, as it has to open the command file
7383  * and set yyin appropriately.  Once that is done, though, it just calls
7384  * read_input(), and only once, since prompting is not possible.
7385  */

7387 static int
7388 cmd_file(char *file)
7389 {
7390        FILE *infile;
7391        int err;
7392        struct stat statbuf;
7393        boolean_t using_real_file = (strcmp(file, "-") != 0);

7395        if (using_real_file) {
7396                /*
7397                 * zerr() prints a line number in cmd_file_mode, which we do
7398                 * not want here, so temporarily unset it.
7399                 */
7400                cmd_file_mode = B_FALSE;
```

```
7401                 if ((infile = fopen(file, "r")) == NULL) {
7402                         zerr(gettext("could not open file %s: %s"),
7403                             file, strerror(errno));
7404                         return (Z_ERR);
7405                 }
7406                 if ((err = fstat(fileno(infile), &statbuf)) != 0) {
7407                         zerr(gettext("could not stat file %s: %s"),
7408                             file, strerror(errno));
7409                         err = Z_ERR;
7410                         goto done;
7411                 }
7412                 if (!S_ISREG(statbuf.st_mode)) {
7413                         zerr(gettext("%s is not a regular file."), file);
7414                         err = Z_ERR;
7415                         goto done;
7416                 }
7417                 yyin = infile;
7418                 cmd_file_mode = B_TRUE;
7419                 ok_to_prompt = B_FALSE;
7420         } else {
7421                 /*
7422                  * "-f -" is essentially the same as interactive mode,
7423                  * so treat it that way.
7424                  */
7425                 interactive_mode = B_TRUE;
7426         }
7427         /* Z_REPEAT is for interactive mode; treat it like Z_ERR here. */
7428         if ((err = read_input()) == Z_REPEAT)
7429                 err = Z_ERR;
7430 done:
7431         if (using_real_file)
7432                 (void) fclose(infile);
7433         return (err);
7434 }

7436 /*
7437  * Since yacc is based on reading from a (FILE *) whereas what we get from
7438  * the command line is in argv format, we need to convert when the user
7439  * gives us commands directly from the command line.  That is done here by
7440  * concatenating the argv list into a space-separated string, writing it
7441  * to a temp file, and rewinding the file so yyin can be set to it.  Then
7442  * we call read_input(), and only once, since prompting about whether to
7443  * continue or quit would make no sense in this context.
7444  */

7446 static int
7447 one_command_at_a_time(int argc, char *argv[])
7448 {
7449         char *command;
7450         size_t len = 2; /* terminal \n\0 */
7451         int i, err;

7453         for (i = 0; i < argc; i++)
7454                 len += strlen(argv[i]) + 1;
7455         if ((command = malloc(len)) == NULL) {
7456                 zone_perror(execname, Z_NOMEM, B_TRUE);
7457                 return (Z_ERR);
7458         }
7459         (void) strlcpy(command, argv[0], len);
7460         for (i = 1; i < argc; i++) {
7461                 (void) strlcat(command, " ", len);
7462                 (void) strlcat(command, argv[i], len);
7463         }
7464         (void) strlcat(command, "\n", len);
7465         err = string_to_yyin(command);
7466         free(command);
```



```
7467         if (err != Z_OK)
7468                 return (err);
7469         while (!feof(yyin))
7470                 yyparse();
7471         return (cleanup());
7472 }

7474 static char *
7475 get_execbasename(char *execfullname)
7476 {
7477         char *last_slash, *execbasename;

7479         /* guard against '/' at end of command invocation */
7480         for (;;) {
7481                 last_slash = strrchr(execfullname, '/');
7482                 if (last_slash == NULL) {
7483                         execbasename = execfullname;
7484                         break;
7485                 } else {
7486                         execbasename = last_slash + 1;
7487                         if (*execbasename == '\0') {
7488                                 *last_slash = '\0';
7489                                 continue;
7490                         }
7491                         break;
7492                 }
7493         }
7494         return (execbasename);
7495 }

7497 int
7498 main(int argc, char *argv[])
7499 {
7500         int err, arg;
7501         struct stat st;

7503         /* This must be before anything goes to stdout. */
7504         setbuf(stdout, NULL);

7506         saw_error = B_FALSE;
7507         cmd_file_mode = B_FALSE;
7508         execname = get_execbasename(argv[0]);

7510         (void) setlocale(LC_ALL, "");
7511         (void) textdomain(TEXT_DOMAIN);

7513         if (getzoneid() != GLOBAL_ZONEID) {
7514                 zerr(gettext("%s can only be run from the global zone."),
7515                     execname);
7516                 exit(Z_ERR);
7517         }

7519         if (argc < 2) {
7520                 usage(B_FALSE, HELP_USAGE | HELP_SUBCMDS);
7521                 exit(Z_USAGE);
7522         }
7523         if (strcmp(argv[1], cmd_to_str(CMD_HELP)) == 0) {
7524                 (void) one_command_at_a_time(argc - 1, &(argv[1]));
7525                 exit(Z_OK);
7526         }

7528         while ((arg = getopt(argc, argv, "?f:R:z:")) != EOF) {
7529                 switch (arg) {
7530                 case '?':
7531                         if (optopt == '?')
7532                                 usage(B_TRUE, HELP_USAGE | HELP_SUBCMDS);
```

```
7533                        else
7534                                usage(B_FALSE, HELP_USAGE);
7535                        exit(Z_USAGE);
7536                        /* NOTREACHED */
7537                case 'f':
7538                        cmd_file_name = optarg;
7539                        cmd_file_mode = B_TRUE;
7540                        break;
7541                case 'R':
7542                        if (*optarg != '/') {
7543                                zerr(gettext("root path must be absolute: %s"),
7544                                    optarg);
7545                                exit(Z_USAGE);
7546                        }
7547                        if (stat(optarg, &st) == -1 || !S_ISDIR(st.st_mode)) {
7548                                zerr(gettext(
7549                                    "root path must be a directory: %s"),
7550                                    optarg);
7551                                exit(Z_USAGE);
7552                        }
7553                        zonecfg_set_root(optarg);
7554                        break;
7555                case 'z':
7556                        if (strcmp(optarg, GLOBAL_ZONENAME) == 0) {
7557                                global_zone = B_TRUE;
7558                        } else if (zonecfg_validate_zonename(optarg) != Z_OK) {
7559                                zone_perror(optarg, Z_BOGUS_ZONE_NAME, B_TRUE);
7560                                usage(B_FALSE, HELP_SYNTAX);
7561                                exit(Z_USAGE);
7562                        }
7563                        (void) strlcpy(zone, optarg, sizeof (zone));
7564                        (void) strlcpy(revert_zone, optarg, sizeof (zone));
7565                        break;
7566                default:
7567                        usage(B_FALSE, HELP_USAGE);
7568                        exit(Z_USAGE);
7569                }
7570        }

7572        if (optind > argc || strcmp(zone, "") == 0) {
7573                usage(B_FALSE, HELP_USAGE);
7574                exit(Z_USAGE);
7575        }

7577        if ((err = zonecfg_access(zone, W_OK)) == Z_OK) {
7578                read_only_mode = B_FALSE;
7579        } else if (err == Z_ACCES) {
7580                read_only_mode = B_TRUE;
7581                /* skip this message in one-off from command line mode */
7582                if (optind == argc)
7583                        (void) fprintf(stderr, gettext("WARNING: you do not "
7584                            "have write access to this zone's configuration "
7585                            "file;\ngoing into read-only mode.\n"));
7586        } else {
7587                fprintf(stderr, "%s: Could not access zone configuration "
7588                    "store: %s\n", execname, zonecfg_strerror(err));
7589                exit(Z_ERR);
7590        }

7592        if ((handle = zonecfg_init_handle()) == NULL) {
7593                zone_perror(execname, Z_NOMEM, B_TRUE);
7594                exit(Z_ERR);
7595        }

7597        /*
7598         * This may get set back to FALSE again in cmd_file() if cmd_file_name
```



```
7599         * is a "real" file as opposed to "-" (i.e. meaning use stdin).
7600         */
7601        if (isatty(STDIN_FILENO))
7602                ok_to_prompt = B_TRUE;
7603        if ((gl = new_GetLine(MAX_LINE_LEN, MAX_CMD_HIST)) == NULL)
7604                exit(Z_ERR);
7605        if (gl_customize_completion(gl, NULL, cmd_cpl_fn) != 0)
7606                exit(Z_ERR);
7607        (void) sigset(SIGINT, SIG_IGN);
7608        if (optind == argc) {
7609                if (!cmd_file_mode)
7610                        err = do_interactive();
7611                else
7612                        err = cmd_file(cmd_file_name);
7613        } else {
7614                err = one_command_at_a_time(argc - optind, &(argv[optind]));
7615        }
7616        zonecfg_fini_handle(handle);
7617        if (brand != NULL)
7618                brand_close(brand);
7619        (void) del_GetLine(gl);
7620        return (err);
7621 }
```

```
**********************************************************
   11425 Thu Jun 30 21:58:42 2016
new/usr/src/man/man3proc/Makefile
Code review comments from pmooney (sundry), and igork (screwups in zonecfg refac
**********************************************************
   1 #
   2 # This file and its contents are supplied under the terms of the
   3 # Common Development and Distribution License ("CDDL"), version 1.0.
   4 # You may only use this file in accordance with the terms of version
   5 # 1.0 of the CDDL.
   6 #
   7 # A full copy of the text of the CDDL should have accompanied this
   8 # source.  A copy of the CDDL is also available via the Internet
   9 # at http://www.illumos.org/license/CDDL.
  10 #

  12 #
  13 # Copyright 2011, Richard Lowe
  14 # Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
  15 # Copyright 2015 Joyent, Inc.
  16 #

  18 include         $(SRC)/Makefile.master

  20 MANSECT=        3proc

  22 MANFILES=                                       \
  23                 Lctlfd.3proc                    \
  24                 Lfree.3proc                     \
  25                 Lgrab_error.3proc               \
  26                 Lgrab.3proc                     \
  27                 Lprochandle.3proc               \
  28                 Lpsinfo.3proc                   \
  29                 Lstate.3proc                    \
  30                 Lstatus.3proc                   \
  31                 Paddr_to_ctf.3proc              \
  32                 Paddr_to_loadobj.3proc          \
  33                 Paddr_to_map.3proc              \
  34                 Pasfd.3proc                     \
  35                 Pclearfault.3proc               \
  36                 Pclearsig.3proc                 \
  37                 Pcontent.3proc                  \
  38                 Pcreate_agent.3proc             \
  39                 Pcreate_error.3proc             \
  40                 Pcreate.3proc                   \
  41                 Pcred.3proc                     \
  42                 Pctlfd.3proc                    \
  43                 Pdelbkpt.3proc                  \
  44                 Pdelwapt.3proc                  \
  45                 Pdestroy_agent.3proc            \
  46                 Penv_iter.3proc                 \
  47                 Perror_printf.3proc             \
  48                 Pexecname.3proc                 \
  49                 Pfault.3proc                    \
  50                 Pfdinfo_iter.3proc              \
  51                 Pgcore.3proc                    \
  52                 Pgetareg.3proc                  \
  53                 Pgetauxval.3proc                \
  54                 Pgetauxvec.3proc                \
  55                 Pgetenv.3proc                   \
  56                 Pgrab_core.3proc                \
  57                 Pgrab_error.3proc               \
  58                 Pgrab_file.3proc                \
  59                 Pgrab.3proc                     \
  60                 Pisprocdir.3proc                \
  61                 Pissyscall.3proc                \
```



```
  62                 Pldt.3proc                      \
  63                 Plmid.3proc                     \
  64                 Plookup_by_addr.3proc           \
  65                 Plwp_getasrs.3proc              \
  66                 Plwp_getgwindows.3proc          \
  67                 Plwp_getpsinfo.3proc            \
  68                 Plwp_getregs.3proc              \
  69                 Plwp_getspymaster.3proc         \
  70                 Plwp_getxregs.3proc             \
  71                 Plwp_iter.3proc                 \
  72                 Plwp_stack.3proc                \
  73                 Pmapping_iter.3proc             \
  74                 Pobjname.3proc                  \
  75                 Pplatform.3proc                 \
  76                 Ppltdest.3proc                  \
  77                 Ppriv.3proc                     \
  78                 Ppsinfo.3proc                   \
  79                 pr_access.3proc                 \
  80                 pr_close.3proc                  \
  81                 pr_creat.3proc                  \
  82                 pr_door_info.3proc              \
  83                 pr_exit.3proc                   \
  84                 pr_fcntl.3proc                  \
  85                 pr_fstatvfs.3proc               \
  86                 pr_getitimer.3proc              \
  87                 pr_getpeername.3proc            \
  88                 pr_getpeerucred.3proc           \
  89                 pr_getprojid.3proc              \
  90                 pr_getrctl.3proc                \
  91                 pr_getrlimit.3proc              \
  92                 pr_getsockname.3proc            \
  93                 pr_getsockopt.3proc             \
  94                 pr_gettaskid.3proc              \
  95                 pr_getzoneid.3proc              \
  96                 pr_ioctl.3proc                  \
  97                 pr_link.3proc                   \
  98                 pr_llseek.3proc                 \
  99                 pr_lseek.3proc                  \
 100                 pr_memcntl.3proc                \
 101                 pr_meminfo.3proc                \
 102                 pr_mmap.3proc                   \
 103                 pr_munmap.3proc                 \
 104                 pr_open.3proc                   \
 105                 pr_processor_bind.3proc         \
 106                 pr_rename.3proc                 \
 107                 pr_setitimer.3proc              \
 108                 pr_setrctl.3proc                \
 109                 pr_setrlimit.3proc              \
 110                 pr_settaskid.3proc              \
 111                 pr_sigaction.3proc              \
 112                 pr_stat.3proc                   \
 113                 pr_statvfs.3proc                \
 114                 pr_unlink.3proc                 \
 115                 pr_waitid.3proc                 \
 116                 Prd_agent.3proc                 \
 117                 Pread.3proc                     \
 118                 Prelease.3proc                  \
 119                 Preopen.3proc                   \
 120                 Preset_maps.3proc               \
 121                 proc_arg_grab.3proc             \
 122                 proc_arg_psinfo.3proc           \
 123                 proc_content2str.3proc          \
 124                 proc_fltname.3proc              \
 125                 proc_fltset2str.3proc           \
 126                 proc_get_auxv.3proc             \
 127                 proc_get_cred.3proc             \
```

```
128                 proc_get_priv.3proc             \
129                 proc_get_psinfo.3proc           \
130                 proc_get_status.3proc           \
131                 proc_initstdio.3proc            \
132                 proc_lwp_in_set.3proc           \
133                 proc_service.3proc              \
134                 proc_str2flt.3proc              \
135                 proc_str2fltset.3proc           \
136                 proc_unctrl_psinfo.3proc        \
137                 proc_walk.3proc                 \
138                 Psecflags.3proc                 \
139 #endif /* ! codereview */
140                 Psetbkpt.3proc                  \
141                 Psetcred.3proc                  \
142                 Psetfault.3proc                 \
143                 Psetflags.3proc                 \
144                 Psetpriv.3proc                  \
145                 Psetrun.3proc                   \
146                 Psetsignal.3proc                \
147                 Psetsysentry.3proc              \
148                 Psetwapt.3proc                  \
149                 Psetzoneid.3proc                \
150                 Psignal.3proc                   \
151                 Pstack_iter.3proc               \
152                 Pstate.3proc                    \
153                 Pstatus.3proc                   \
154                 Pstopstatus.3proc               \
155                 Psymbol_iter.3proc              \
156                 Psync.3proc                     \
157                 Psysentry.3proc                 \
158                 Puname.3proc                    \
159                 Pupdate_maps.3proc              \
160                 Pupdate_syms.3proc              \
161                 Pwrite.3proc                    \
162                 Pxecbkpt.3proc                  \
163                 Pzonename.3proc                 \
164                 ps_lgetregs.3proc               \
165                 ps_pglobal_lookup.3proc         \
166                 ps_pread.3proc                  \
167                 ps_pstop.3proc


170 MANLINKS=                                       \
171                 Lalt_stack.3proc                \
172                 Lclearfault.3proc               \
173                 Lclearsig.3proc                 \
174                 Ldstop.3proc                    \
175                 Lgetareg.3proc                  \
176                 Lmain_stack.3proc               \
177                 Lputareg.3proc                  \
178                 Lsetrun.3proc                   \
179                 Lstack.3proc                    \
180                 Lstop.3proc                     \
181                 Lsync.3proc                     \
182                 Lwait.3proc                     \
183                 Lxecbkpt.3proc                  \
184                 Lxecwapt.3proc                  \
185                 Paddr_to_text_map.3proc         \
186                 Pcreate_callback.3proc          \
187                 Pdstop.3proc                    \
188                 Pfgcore.3proc                   \
189                 Pfgrab_core.3proc               \
190                 Pfree.3proc                     \
191                 Pissyscall_prev.3proc           \
192                 Plmid_to_ctf.3proc              \
193                 Plmid_to_loadobj.3proc          \
```



```
194                 Plmid_to_map.3proc              \
195                 Plookup_by_name.3proc           \
196                 Plwp_alt_stack.3proc            \
197                 Plwp_getfpregs.3proc            \
198                 Plwp_iter_all.3proc             \
199                 Plwp_main_stack.3proc           \
200                 Plwp_setasrs.3proc              \
201                 Plwp_setfpregs.3proc            \
202                 Plwp_setregs.3proc              \
203                 Plwp_setxregs.3proc             \
204                 Pmapping_iter_resolved.3proc    \
205                 Pname_to_ctf.3proc              \
206                 Pname_to_loadobj.3proc          \
207                 Pname_to_map.3proc              \
208                 Pobject_iter_resolved.3proc     \
209                 Pobject_iter.3proc              \
210                 Pobjname_resolved.3proc         \
211                 Ppriv_free.3proc                \
212                 Pputareg.3proc                  \
213                 pr_fstat.3proc                  \
214                 pr_fstat64.3proc                \
215                 pr_getrlimit64.3proc            \
216                 pr_lstat.3proc                  \
217                 pr_lstat64.3proc                \
218                 pr_setrlimit64.3proc            \
219                 pr_stat64.3proc                 \
220                 Pread_string.3proc              \
221                 proc_arg_xgrab.3proc            \
222                 proc_arg_xpsinfo.3proc          \
223                 proc_finistdio.3proc            \
224                 proc_flushstdio.3proc           \
225                 proc_free_priv.3proc            \
226                 proc_get_ldt.3proc              \
227                 proc_lwp_range_valid.3proc      \
228                 proc_signame.3proc              \
229                 proc_sigset2str.3proc           \
230                 proc_str2content.3proc          \
231                 proc_str2sig.3proc              \
232                 proc_str2sigset.3proc           \
233                 proc_str2sys.3proc              \
234                 proc_str2sysset.3proc           \
235                 proc_sysname.3proc              \
236                 proc_sysset2str.3proc           \
237                 ps_kill.3proc                   \
238                 ps_lcontinue.3proc              \
239                 ps_lgetfpregs.3proc             \
240                 ps_lgetxregs.3proc              \
241                 ps_lgetxregsize.3proc           \
242                 ps_lrolltoaddr.3proc            \
243                 ps_lsetfpregs.3proc             \
244                 ps_lsetregs.3proc               \
245                 ps_lsetxregs.3proc              \
246                 ps_lstop.3proc                  \
247                 ps_pcontinue.3proc              \
248                 ps_pdread.3proc                 \
249                 ps_pdwrite.3proc                \
250                 ps_pglobal_sym.3proc            \
251                 ps_ptread.3proc                 \
252                 ps_ptwrite.3proc                \
253                 ps_pwrite.3proc                 \
254                 Psetsysexit.3proc               \
255                 Pstop.3proc                     \
256                 Psymbol_iter_by_addr.3proc      \
257                 Psymbol_iter_by_lmid.3proc      \
258                 Psymbol_iter_by_name.3proc      \
259                 Psysexit.3proc                  \
```

```
260                 Punsetflags.3proc               \
261                 Pwait.3proc                     \
262                 Pxcreate.3proc                  \
263                 Pxecwapt.3proc                  \
264                 Pxlookup_by_addr_resolved.3proc \
265                 Pxlookup_by_addr.3proc          \
266                 Pxlookup_by_name.3proc          \
267                 Pxsymbol_iter.3proc             \
268                 Pzonepath.3proc                 \
269                 Pzoneroot.3proc


272 ps_lgetfpregs.3proc             := LINKSRC = ps_lgetregs.3proc
273 ps_lgetxregs.3proc              := LINKSRC = ps_lgetregs.3proc
274 ps_lgetxregsize.3proc           := LINKSRC = ps_lgetregs.3proc
275 ps_lsetfpregs.3proc             := LINKSRC = ps_lgetregs.3proc
276 ps_lsetregs.3proc               := LINKSRC = ps_lgetregs.3proc
277 ps_lsetxregs.3proc              := LINKSRC = ps_lgetregs.3proc

279 ps_pglobal_sym.3proc           := LINKSRC = ps_pglobal_lookup.3proc

281 ps_pdread.3proc                := LINKSRC = ps_pread.3proc
282 ps_pdwrite.3proc               := LINKSRC = ps_pread.3proc
283 ps_ptread.3proc                := LINKSRC = ps_pread.3proc
284 ps_ptwrite.3proc               := LINKSRC = ps_pread.3proc
285 ps_pwrite.3proc                := LINKSRC = ps_pread.3proc

287 ps_kill.3proc                  := LINKSRC = ps_pstop.3proc
288 ps_lcontinue.3proc             := LINKSRC = ps_pstop.3proc
289 ps_lrolltoaddr.3proc           := LINKSRC = ps_pstop.3proc
290 ps_lstop.3proc                 := LINKSRC = ps_pstop.3proc
291 ps_pcontinue.3proc             := LINKSRC = ps_pstop.3proc


294 Pxcreate.3proc                 := LINKSRC = Pcreate.3proc
295 Pcreate_callback.3proc         := LINKSRC = Pcreate.3proc

297 Pfgrab_core.3proc             := LINKSRC = Pgrab_core.3proc

299 Pfree.3proc                    := LINKSRC = Prelease.3proc

301 Plwp_iter_all.3proc           := LINKSRC = Plwp_iter.3proc

303 Pmapping_iter_resolved.3proc  := LINKSRC = Pmapping_iter.3proc
304 Pobject_iter.3proc            := LINKSRC = Pmapping_iter.3proc
305 Pobject_iter_resolved.3proc   := LINKSRC = Pmapping_iter.3proc

307 Psymbol_iter_by_addr.3proc    := LINKSRC = Psymbol_iter.3proc
308 Psymbol_iter_by_lmid.3proc    := LINKSRC = Psymbol_iter.3proc
309 Psymbol_iter_by_name.3proc    := LINKSRC = Psymbol_iter.3proc
310 Pxsymbol_iter.3proc           := LINKSRC = Psymbol_iter.3proc

312 Plmid_to_ctf.3proc            := LINKSRC = Paddr_to_ctf.3proc
313 Pname_to_ctf.3proc            := LINKSRC = Paddr_to_ctf.3proc

315 Plmid_to_loadobj.3proc       := LINKSRC = Paddr_to_loadobj.3proc
316 Pname_to_loadobj.3proc       := LINKSRC = Paddr_to_loadobj.3proc

318 Paddr_to_text_map.3proc      := LINKSRC = Paddr_to_map.3proc
319 Plmid_to_map.3proc           := LINKSRC = Paddr_to_map.3proc
320 Pname_to_map.3proc           := LINKSRC = Paddr_to_map.3proc

322 Pdstop.3proc                 := LINKSRC = Pstopstatus.3proc
323 Pstop.3proc                  := LINKSRC = Pstopstatus.3proc
324 Pwait.3proc                  := LINKSRC = Pstopstatus.3proc
325 Ldstop.3proc                 := LINKSRC = Pstopstatus.3proc
```



```
326 Lstop.3proc                  := LINKSRC = Pstopstatus.3proc
327 Lwait.3proc                  := LINKSRC = Pstopstatus.3proc

329 Lsync.3proc                  := LINKSRC = Psync.3proc

331 Pfgcore.3proc                := LINKSRC = Pgcore.3proc

333 Pputareg.3proc               := LINKSRC = Pgetareg.3proc
334 Lgetareg.3proc               := LINKSRC = Pgetareg.3proc
335 Lputareg.3proc               := LINKSRC = Pgetareg.3proc

337 Pissyscall_prev.3proc        := LINKSRC = Pissyscall.3proc

339 Pxlookup_by_addr.3proc       := LINKSRC = Plookup_by_addr.3proc
340 Pxlookup_by_addr_resolved.3proc := LINKSRC = Plookup_by_addr.3proc
341 Plookup_by_name.3proc        := LINKSRC = Plookup_by_addr.3proc
342 Pxlookup_by_name.3proc       := LINKSRC = Plookup_by_addr.3proc

344 Plwp_setregs.3proc          := LINKSRC = Plwp_getregs.3proc
345 Plwp_getfpregs.3proc        := LINKSRC = Plwp_getregs.3proc
346 Plwp_setfpregs.3proc        := LINKSRC = Plwp_getregs.3proc

348 Plwp_alt_stack.3proc        := LINKSRC = Plwp_stack.3proc
349 Plwp_main_stack.3proc       := LINKSRC = Plwp_stack.3proc
350 Lalt_stack.3proc            := LINKSRC = Plwp_stack.3proc
351 Lmain_stack.3proc           := LINKSRC = Plwp_stack.3proc
352 Lstack.3proc                := LINKSRC = Plwp_stack.3proc

354 Pobjname_resolved.3proc     := LINKSRC = Pobjname.3proc

356 Ppriv_free.3proc            := LINKSRC = Ppriv.3proc

358 Pread_string.3proc          := LINKSRC = Pread.3proc

360 Punsetflags.3proc          := LINKSRC = Psetflags.3proc

362 Psetsysexit.3proc          := LINKSRC = Psetsysentry.3proc

364 Psysexit.3proc             := LINKSRC = Psysentry.3proc

366 Pxecwapt.3proc             := LINKSRC = Pxecbkpt.3proc
367 Lxecbkpt.3proc             := LINKSRC = Pxecbkpt.3proc
368 Lxecwapt.3proc             := LINKSRC = Pxecbkpt.3proc

370 Lclearfault.3proc          := LINKSRC = Pclearfault.3proc

372 Lclearsig.3proc           := LINKSRC = Pclearsig.3proc

374 Lsetrun.3proc             := LINKSRC = Psetrun.3proc

376 Pzonepath.3proc           := LINKSRC = Pzonename.3proc
377 Pzoneroot.3proc           := LINKSRC = Pzonename.3proc

379 pr_fstat.3proc            := LINKSRC = pr_stat.3proc
380 pr_fstat64.3proc          := LINKSRC = pr_stat.3proc
381 pr_lstat.3proc            := LINKSRC = pr_stat.3proc
382 pr_lstat64.3proc          := LINKSRC = pr_stat.3proc
383 pr_stat64.3proc           := LINKSRC = pr_stat.3proc

385 pr_getrlimit64.3proc      := LINKSRC = pr_getrlimit.3proc

387 pr_setrlimit64.3proc      := LINKSRC = pr_setrlimit.3proc

389 proc_arg_xgrab.3proc      := LINKSRC = proc_arg_grab.3proc

391 proc_arg_xpsinfo.3proc    := LINKSRC = proc_arg_psinfo.3proc
```

```
393 proc_str2content.3proc          := LINKSRC = proc_content2str.3proc

395 proc_flushstdio.3proc           := LINKSRC = proc_initstdio.3proc
396 proc_finistdio.3proc            := LINKSRC = proc_initstdio.3proc

398 proc_signame.3proc              := LINKSRC = proc_fltname.3proc
399 proc_sysname.3proc              := LINKSRC = proc_fltname.3proc

401 proc_sigset2str.3proc           := LINKSRC = proc_fltset2str.3proc
402 proc_sysset2str.3proc           := LINKSRC = proc_fltset2str.3proc

404 proc_free_priv.3proc            := LINKSRC = proc_get_priv.3proc

406 proc_lwp_range_valid.3proc      := LINKSRC = proc_lwp_in_set.3proc

408 proc_str2sig.3proc              := LINKSRC = proc_str2flt.3proc
409 proc_str2sys.3proc              := LINKSRC = proc_str2flt.3proc

411 proc_str2sigset.3proc           := LINKSRC = proc_str2fltset.3proc
412 proc_str2sysset.3proc           := LINKSRC = proc_str2fltset.3proc

414 proc_get_ldt.3proc              := LINKSRC = Pldt.3proc

416 Plwp_setxregs.3proc             := LINKSRC = Plwp_getxregs.3proc

418 Plwp_setasrs.3proc             := LINKSRC = Plwp_getasrs.3proc

420 .KEEP_STATE:

422 include         $(SRC)/man/Makefile.man

424 install:        $(ROOTMANFILES) $(ROOTMANLINKS)
```

```
***********************************************************
   13704 Thu Jun 30 21:58:43 2016
new/usr/src/pkg/manifests/system-library.man3proc.inc
Code review comments from pmooney (sundry), and igork (screwups in zonecfg refac
***********************************************************
```

```
   1 #
   2 # This file and its contents are supplied under the terms of the
   3 # Common Development and Distribution License ("CDDL"), version 1.0.
   4 # You may only use this file in accordance with the terms of version
   5 # 1.0 of the CDDL.
   6 #
   7 # A full copy of the text of the CDDL should have accompanied this
   8 # source.  A copy of the CDDL is also available via the Internet
   9 # at http://www.illumos.org/license/CDDL.
  10 #

  12 #
  13 # Copyright 2015 Joyent, Inc.
  14 #


  17 file path=usr/share/man/man3proc/Lctlfd.3proc
  18 file path=usr/share/man/man3proc/Lfree.3proc
  19 file path=usr/share/man/man3proc/Lgrab_error.3proc
  20 file path=usr/share/man/man3proc/Lgrab.3proc
  21 file path=usr/share/man/man3proc/Lprochandle.3proc
  22 file path=usr/share/man/man3proc/Lpsinfo.3proc
  23 file path=usr/share/man/man3proc/Lstate.3proc
  24 file path=usr/share/man/man3proc/Lstatus.3proc
  25 file path=usr/share/man/man3proc/Paddr_to_ctf.3proc
  26 file path=usr/share/man/man3proc/Paddr_to_loadobj.3proc
  27 file path=usr/share/man/man3proc/Paddr_to_map.3proc
  28 file path=usr/share/man/man3proc/Pasfd.3proc
  29 file path=usr/share/man/man3proc/Pclearfault.3proc
  30 file path=usr/share/man/man3proc/Pclearsig.3proc
  31 file path=usr/share/man/man3proc/Pcontent.3proc
  32 file path=usr/share/man/man3proc/Pcreate_agent.3proc
  33 file path=usr/share/man/man3proc/Pcreate_error.3proc
  34 file path=usr/share/man/man3proc/Pcreate.3proc
  35 file path=usr/share/man/man3proc/Pcred.3proc
  36 file path=usr/share/man/man3proc/Pctlfd.3proc
  37 file path=usr/share/man/man3proc/Pdelbkpt.3proc
  38 file path=usr/share/man/man3proc/Pdelwapt.3proc
  39 file path=usr/share/man/man3proc/Pdestroy_agent.3proc
  40 file path=usr/share/man/man3proc/Penv_iter.3proc
  41 file path=usr/share/man/man3proc/Perror_printf.3proc
  42 file path=usr/share/man/man3proc/Pexecname.3proc
  43 file path=usr/share/man/man3proc/Pfault.3proc
  44 file path=usr/share/man/man3proc/Pfdinfo_iter.3proc
  45 file path=usr/share/man/man3proc/Pgcore.3proc
  46 file path=usr/share/man/man3proc/Pgetareg.3proc
  47 file path=usr/share/man/man3proc/Pgetauxval.3proc
  48 file path=usr/share/man/man3proc/Pgetauxvec.3proc
  49 file path=usr/share/man/man3proc/Pgetenv.3proc
  50 file path=usr/share/man/man3proc/Pgrab_core.3proc
  51 file path=usr/share/man/man3proc/Pgrab_error.3proc
  52 file path=usr/share/man/man3proc/Pgrab_file.3proc
  53 file path=usr/share/man/man3proc/Pgrab.3proc
  54 file path=usr/share/man/man3proc/Pisprocdir.3proc
  55 file path=usr/share/man/man3proc/Pissyscall.3proc
  56 file path=usr/share/man/man3proc/Pldt.3proc
  57 file path=usr/share/man/man3proc/Plmid.3proc
  58 file path=usr/share/man/man3proc/Plookup_by_addr.3proc
  59 file path=usr/share/man/man3proc/Plwp_getasrs.3proc
  60 file path=usr/share/man/man3proc/Plwp_getgwindows.3proc
  61 file path=usr/share/man/man3proc/Plwp_getpsinfo.3proc
```



```
  62 file path=usr/share/man/man3proc/Plwp_getregs.3proc
  63 file path=usr/share/man/man3proc/Plwp_getspymaster.3proc
  64 file path=usr/share/man/man3proc/Plwp_getxregs.3proc
  65 file path=usr/share/man/man3proc/Plwp_iter.3proc
  66 file path=usr/share/man/man3proc/Plwp_stack.3proc
  67 file path=usr/share/man/man3proc/Pmapping_iter.3proc
  68 file path=usr/share/man/man3proc/Pobjname.3proc
  69 file path=usr/share/man/man3proc/Pplatform.3proc
  70 file path=usr/share/man/man3proc/Ppltdest.3proc
  71 file path=usr/share/man/man3proc/Ppriv.3proc
  72 file path=usr/share/man/man3proc/Ppsinfo.3proc
  73 file path=usr/share/man/man3proc/pr_access.3proc
  74 file path=usr/share/man/man3proc/pr_close.3proc
  75 file path=usr/share/man/man3proc/pr_creat.3proc
  76 file path=usr/share/man/man3proc/pr_door_info.3proc
  77 file path=usr/share/man/man3proc/pr_exit.3proc
  78 file path=usr/share/man/man3proc/pr_fcntl.3proc
  79 file path=usr/share/man/man3proc/pr_fstatvfs.3proc
  80 file path=usr/share/man/man3proc/pr_getitimer.3proc
  81 file path=usr/share/man/man3proc/pr_getpeername.3proc
  82 file path=usr/share/man/man3proc/pr_getpeerucred.3proc
  83 file path=usr/share/man/man3proc/pr_getprojid.3proc
  84 file path=usr/share/man/man3proc/pr_getrctl.3proc
  85 file path=usr/share/man/man3proc/pr_getrlimit.3proc
  86 file path=usr/share/man/man3proc/pr_getsockname.3proc
  87 file path=usr/share/man/man3proc/pr_getsockopt.3proc
  88 file path=usr/share/man/man3proc/pr_gettaskid.3proc
  89 file path=usr/share/man/man3proc/pr_getzoneid.3proc
  90 file path=usr/share/man/man3proc/pr_ioctl.3proc
  91 file path=usr/share/man/man3proc/pr_link.3proc
  92 file path=usr/share/man/man3proc/pr_llseek.3proc
  93 file path=usr/share/man/man3proc/pr_lseek.3proc
  94 file path=usr/share/man/man3proc/pr_memcntl.3proc
  95 file path=usr/share/man/man3proc/pr_meminfo.3proc
  96 file path=usr/share/man/man3proc/pr_mmap.3proc
  97 file path=usr/share/man/man3proc/pr_munmap.3proc
  98 file path=usr/share/man/man3proc/pr_open.3proc
  99 file path=usr/share/man/man3proc/pr_processor_bind.3proc
 100 file path=usr/share/man/man3proc/pr_rename.3proc
 101 file path=usr/share/man/man3proc/pr_setitimer.3proc
 102 file path=usr/share/man/man3proc/pr_setrctl.3proc
 103 file path=usr/share/man/man3proc/pr_setrlimit.3proc
 104 file path=usr/share/man/man3proc/pr_settaskid.3proc
 105 file path=usr/share/man/man3proc/pr_sigaction.3proc
 106 file path=usr/share/man/man3proc/pr_stat.3proc
 107 file path=usr/share/man/man3proc/pr_statvfs.3proc
 108 file path=usr/share/man/man3proc/pr_unlink.3proc
 109 file path=usr/share/man/man3proc/pr_waitid.3proc
 110 file path=usr/share/man/man3proc/Prd_agent.3proc
 111 file path=usr/share/man/man3proc/Pread.3proc
 112 file path=usr/share/man/man3proc/Prelease.3proc
 113 file path=usr/share/man/man3proc/Preopen.3proc
 114 file path=usr/share/man/man3proc/Preset_maps.3proc
 115 file path=usr/share/man/man3proc/proc_arg_grab.3proc
 116 file path=usr/share/man/man3proc/proc_arg_psinfo.3proc
 117 file path=usr/share/man/man3proc/proc_content2str.3proc
 118 file path=usr/share/man/man3proc/proc_fltname.3proc
 119 file path=usr/share/man/man3proc/proc_fltset2str.3proc
 120 file path=usr/share/man/man3proc/proc_get_auxv.3proc
 121 file path=usr/share/man/man3proc/proc_get_cred.3proc
 122 file path=usr/share/man/man3proc/proc_get_priv.3proc
 123 file path=usr/share/man/man3proc/proc_get_psinfo.3proc
 124 file path=usr/share/man/man3proc/proc_get_status.3proc
 125 file path=usr/share/man/man3proc/proc_initstdio.3proc
 126 file path=usr/share/man/man3proc/proc_lwp_in_set.3proc
 127 file path=usr/share/man/man3proc/proc_str2flt.3proc
```

```
128 file path=usr/share/man/man3proc/proc_str2fltset.3proc
129 file path=usr/share/man/man3proc/proc_unctrl_psinfo.3proc
130 file path=usr/share/man/man3proc/proc_walk.3proc
131 file path=usr/share/man/man3proc/Psecflags.3proc
132 #endif /* ! codereview */
133 file path=usr/share/man/man3proc/Psetbkpt.3proc
134 file path=usr/share/man/man3proc/Psetcred.3proc
135 file path=usr/share/man/man3proc/Psetfault.3proc
136 file path=usr/share/man/man3proc/Psetflags.3proc
137 file path=usr/share/man/man3proc/Psetpriv.3proc
138 file path=usr/share/man/man3proc/Psetrun.3proc
139 file path=usr/share/man/man3proc/Psetsignal.3proc
140 file path=usr/share/man/man3proc/Psetsysentry.3proc
141 file path=usr/share/man/man3proc/Psetwapt.3proc
142 file path=usr/share/man/man3proc/Psetzoneid.3proc
143 file path=usr/share/man/man3proc/Psignal.3proc
144 file path=usr/share/man/man3proc/Pstack_iter.3proc
145 file path=usr/share/man/man3proc/Pstate.3proc
146 file path=usr/share/man/man3proc/Pstatus.3proc
147 file path=usr/share/man/man3proc/Pstopstatus.3proc
148 file path=usr/share/man/man3proc/Psymbol_iter.3proc
149 file path=usr/share/man/man3proc/Psync.3proc
150 file path=usr/share/man/man3proc/Psysentry.3proc
151 file path=usr/share/man/man3proc/Puname.3proc
152 file path=usr/share/man/man3proc/Pupdate_maps.3proc
153 file path=usr/share/man/man3proc/Pupdate_syms.3proc
154 file path=usr/share/man/man3proc/Pwrite.3proc
155 file path=usr/share/man/man3proc/Pxecbkpt.3proc
156 file path=usr/share/man/man3proc/Pzonename.3proc
157 link path=usr/share/man/man3proc/Lalt_stack.3proc target=Plwp_stack.3proc
158 link path=usr/share/man/man3proc/Lclearfault.3proc target=Pclearfault.3proc
159 link path=usr/share/man/man3proc/Lclearsig.3proc target=Pclearsig.3proc
160 link path=usr/share/man/man3proc/Ldstop.3proc target=Pstopstatus.3proc
161 link path=usr/share/man/man3proc/Lgetareg.3proc target=Pgetareg.3proc
162 link path=usr/share/man/man3proc/Lmain_stack.3proc target=Plwp_stack.3proc
163 link path=usr/share/man/man3proc/Lputareg.3proc target=Pgetareg.3proc
164 link path=usr/share/man/man3proc/Lsetrun.3proc target=Psetrun.3proc
165 link path=usr/share/man/man3proc/Lstack.3proc target=Plwp_stack.3proc
166 link path=usr/share/man/man3proc/Lstop.3proc target=Pstopstatus.3proc
167 link path=usr/share/man/man3proc/Lsync.3proc target=Psync.3proc
168 link path=usr/share/man/man3proc/Lwait.3proc target=Pstopstatus.3proc
169 link path=usr/share/man/man3proc/Lxecbkpt.3proc target=Pxecbkpt.3proc
170 link path=usr/share/man/man3proc/Lxecwapt.3proc target=Pxecbkpt.3proc
171 link path=usr/share/man/man3proc/Paddr_to_text_map.3proc target=Paddr_to_map.3pr
172 link path=usr/share/man/man3proc/Pcreate_callback.3proc target=Pcreate.3proc
173 link path=usr/share/man/man3proc/Pdstop.3proc target=Pstopstatus.3proc
174 link path=usr/share/man/man3proc/Pfgcore.3proc target=Pgcore.3proc
175 link path=usr/share/man/man3proc/Pfgrab_core.3proc target=Pgrab_core.3proc
176 link path=usr/share/man/man3proc/Pfree.3proc target=Prelease.3proc
177 link path=usr/share/man/man3proc/Pissyscall_prev.3proc target=Pissyscall.3proc
178 link path=usr/share/man/man3proc/Plmid_to_ctf.3proc target=Paddr_to_ctf.3proc
179 link path=usr/share/man/man3proc/Plmid_to_loadobj.3proc target=Paddr_to_loadobj.
180 link path=usr/share/man/man3proc/Plmid_to_map.3proc target=Paddr_to_map.3proc
181 link path=usr/share/man/man3proc/Plookup_by_name.3proc target=Plookup_by_addr.3p
182 link path=usr/share/man/man3proc/Plwp_alt_stack.3proc target=Plwp_stack.3proc
183 link path=usr/share/man/man3proc/Plwp_getfpregs.3proc target=Plwp_getregs.3proc
184 link path=usr/share/man/man3proc/Plwp_iter_all.3proc target=Plwp_iter.3proc
185 link path=usr/share/man/man3proc/Plwp_main_stack.3proc target=Plwp_stack.3proc
186 link path=usr/share/man/man3proc/Plwp_setasrs.3proc target=Plwp_getasrs.3proc
187 link path=usr/share/man/man3proc/Plwp_setfpregs.3proc target=Plwp_getregs.3proc
188 link path=usr/share/man/man3proc/Plwp_setregs.3proc target=Plwp_getregs.3proc
189 link path=usr/share/man/man3proc/Plwp_setxregs.3proc target=Plwp_getxregs.3proc
190 link path=usr/share/man/man3proc/Pmapping_iter_resolved.3proc target=Pmapping_it
191 link path=usr/share/man/man3proc/Pname_to_ctf.3proc target=Paddr_to_ctf.3proc
192 link path=usr/share/man/man3proc/Pname_to_loadobj.3proc target=Paddr_to_loadobj.
193 link path=usr/share/man/man3proc/Pname_to_map.3proc target=Paddr_to_map.3proc
```



```
194 link path=usr/share/man/man3proc/Pobject_iter_resolved.3proc target=Pmapping_ite
195 link path=usr/share/man/man3proc/Pobject_iter.3proc target=Pmapping_iter.3proc
196 link path=usr/share/man/man3proc/Pobjname_resolved.3proc target=Pobjname.3proc
197 link path=usr/share/man/man3proc/Ppriv_free.3proc target=Ppriv.3proc
198 link path=usr/share/man/man3proc/Pputareg.3proc target=Pgetareg.3proc
199 link path=usr/share/man/man3proc/pr_fstat.3proc target=pr_stat.3proc
200 link path=usr/share/man/man3proc/pr_fstat64.3proc target=pr_stat.3proc
201 link path=usr/share/man/man3proc/pr_getrlimit64.3proc target=pr_getrlimit.3proc
202 link path=usr/share/man/man3proc/pr_lstat.3proc target=pr_stat.3proc
203 link path=usr/share/man/man3proc/pr_lstat64.3proc target=pr_stat.3proc
204 link path=usr/share/man/man3proc/pr_setrlimit64.3proc target=pr_setrlimit.3proc
205 link path=usr/share/man/man3proc/pr_stat64.3proc target=pr_stat.3proc
206 link path=usr/share/man/man3proc/Pread_string.3proc target=Pread.3proc
207 link path=usr/share/man/man3proc/proc_arg_xgrab.3proc target=proc_arg_grab.3proc
208 link path=usr/share/man/man3proc/proc_arg_xpsinfo.3proc target=proc_arg_psinfo.3
209 link path=usr/share/man/man3proc/proc_finistdio.3proc target=proc_initstdio.3pro
210 link path=usr/share/man/man3proc/proc_flushstdio.3proc target=proc_initstdio.3pr
211 link path=usr/share/man/man3proc/proc_free_priv.3proc target=proc_get_priv.3proc
212 link path=usr/share/man/man3proc/proc_get_ldt.3proc target=Pldt.3proc
213 link path=usr/share/man/man3proc/proc_lwp_range_valid.3proc target=proc_lwp_in_s
214 link path=usr/share/man/man3proc/proc_signame.3proc target=proc_fltname.3proc
215 link path=usr/share/man/man3proc/proc_sigset2str.3proc target=proc_fltset2str.3p
216 link path=usr/share/man/man3proc/proc_str2content.3proc target=proc_content2str.
217 link path=usr/share/man/man3proc/proc_str2sig.3proc target=proc_str2flt.3proc
218 link path=usr/share/man/man3proc/proc_str2sigset.3proc target=proc_str2fltset.3p
219 link path=usr/share/man/man3proc/proc_str2sys.3proc target=proc_str2flt.3proc
220 link path=usr/share/man/man3proc/proc_str2sysset.3proc target=proc_str2fltset.3p
221 link path=usr/share/man/man3proc/proc_sysname.3proc target=proc_fltname.3proc
222 link path=usr/share/man/man3proc/proc_sysset2str.3proc target=proc_fltset2str.3p
223 link path=usr/share/man/man3proc/Psetsysexit.3proc target=Psetsysentry.3proc
224 link path=usr/share/man/man3proc/Pstop.3proc target=Pstopstatus.3proc
225 link path=usr/share/man/man3proc/Psymbol_iter_by_addr.3proc target=Psymbol_iter.
226 link path=usr/share/man/man3proc/Psymbol_iter_by_lmid.3proc target=Psymbol_iter.
227 link path=usr/share/man/man3proc/Psymbol_iter_by_name.3proc target=Psymbol_iter.
228 link path=usr/share/man/man3proc/Psysexit.3proc target=Psysentry.3proc
229 link path=usr/share/man/man3proc/Punsetflags.3proc target=Psetflags.3proc
230 link path=usr/share/man/man3proc/Pwait.3proc target=Pstopstatus.3proc
231 link path=usr/share/man/man3proc/Pxcreate.3proc target=Pcreate.3proc
232 link path=usr/share/man/man3proc/Pxecwapt.3proc target=Pxecbkpt.3proc
233 link path=usr/share/man/man3proc/Pxlookup_by_addr_resolved.3proc target=Plookup_
234 link path=usr/share/man/man3proc/Pxlookup_by_addr.3proc target=Plookup_by_addr.3
235 link path=usr/share/man/man3proc/Pxlookup_by_name.3proc target=Plookup_by_addr.3
236 link path=usr/share/man/man3proc/Pxsymbol_iter.3proc target=Psymbol_iter.3proc
237 link path=usr/share/man/man3proc/Pzonepath.3proc target=Pzonename.3proc
238 link path=usr/share/man/man3proc/Pzoneroot.3proc target=Pzonename.3proc
```

```
**********************************************************
    3782 Thu Jun 30 21:58:44 2016
new/usr/src/test/os-tests/tests/secflags/secflags_zonecfg.sh
Code review comments from pmooney (sundry), and igork (screwups in zonecfg refac
**********************************************************
______unchanged_portion_omitted_

  84 ret=0

  86 expect_success valid-no-config <<EOF
  87 EOF
  88 (( $? != 0 )) && ret=1

  90 #endif /* ! codereview */
  91 expect_success valid-full-config <<EOF
  92 add security-flags
  93 set lower=none
  94 set default=aslr
  95 set upper=all
  96 end
  97 EOF
  98 (( $? != 0 )) && ret=1

 100 expect_success valid-partial-config <<EOF
 101 add security-flags
 102 set default=aslr
 103 end
 104 EOF
 105 (( $? != 0 )) && ret=1

 107 expect_fail invalid-full-lower-gt-def "default secflags must be above the lower
 108 add security-flags
 109 set lower=aslr
 110 set default=none
 111 set upper=all
 112 end
 113 EOF
 114 (( $? != 0 )) && ret=1

 116 expect_fail invalid-partial-lower-gt-def "default secflags must be above the low
 117 add security-flags
 118 set lower=aslr
 119 set default=none
 120 end
 121 EOF
 122 (( $? != 0 )) && ret=1

 124 expect_fail invalid-full-def-gt-upper "default secflags must be within the upper
 125 add security-flags
 126 set lower=none
 127 set default=all
 128 set upper=none
 129 end
 130 EOF
 131 (( $? != 0 )) && ret=1

 133 expect_fail invalid-partial-def-gt-upper "default secflags must be within the up
 134 add security-flags
 135 set default=all
 136 set upper=none
 137 end
 138 EOF
 139 (( $? != 0 )) && ret=1

 141 expect_fail invalid-full-def-gt-upper "default secflags must be within the upper
 142 add security-flags
```



```
 143 set lower=none
 144 set default=all
 145 set upper=none
 146 end
 147 EOF
 148 (( $? != 0 )) && ret=1

 150 expect_fail invalid-partial-lower-gt-upper "lower secflags must be within the up
 151 add security-flags
 152 set lower=all
 153 set upper=none
 154 end
 155 EOF
 156 (( $? != 0 )) && ret=1

 158 expect_fail invalid-parse-fail-def "default security flags 'fail' are invalid" <
 159 add security-flags
 160 set default=fail
 161 end
 162 EOF
 163 (( $? != 0 )) && ret=1

 165 expect_fail invalid-parse-fail-lower "lower security flags 'fail' are invalid" <
 166 add security-flags
 167 set lower=fail
 168 end
 169 EOF
 170 (( $? != 0 )) && ret=1

 172 expect_fail invalid-parse-fail-def "upper security flags 'fail' are invalid" <<E
 173 add security-flags
 174 set upper=fail
 175 end
 176 EOF
 177 (( $? != 0 )) && ret=1

 179 exit $ret
```

```
**********************************************************
   46455 Thu Jun 30 21:58:44 2016
new/usr/src/uts/common/os/sysent.c
Code review comments from pmooney (sundry), and igork (screwups in zonecfg refac
**********************************************************
```

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

/*
 * Copyright (c) 1988, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2012 Milan Jurik. All rights reserved.
 * Copyright (c) 2013, OmniTI Computer Consulting, Inc. All rights reserved.
 * Copyright (c) 2015, Joyent, Inc.
 */

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

#include <sys/param.h>
#include <sys/types.h>
#include <sys/systm.h>
#include <sys/systrace.h>
#include <sys/procfs.h>
#include <sys/mman.h>
#include <sys/int_types.h>
#include <c2/audit.h>
#include <sys/stat.h>
#include <sys/times.h>
#include <sys/statfs.h>
#include <sys/stropts.h>
#include <sys/statvfs.h>
#include <sys/utsname.h>
#include <sys/timex.h>
#include <sys/socket.h>
#include <sys/sendfile.h>

struct hrtsysa;
struct mmaplf32a;

/*
 * This table is the switch used to transfer to the appropriate
 * routine for processing a system call.  Each row contains the
 * number of arguments expected, a switch that tells systrap()
 * in trap.c whether a setjmp() is not necessary, and a pointer
 * to the routine.
 */

int     access(char *, int);
```



```
int     alarm(int);
int     auditsys(struct auditcalls *, rval_t *);
int64_t brandsys(int, uintptr_t, uintptr_t, uintptr_t, uintptr_t, uintptr_t,
    uintptr_t);
intptr_t        brk(caddr_t);
int     brk(caddr_t);
int     chdir(char *);
int     chmod(char *, int);
int     chown(char *, uid_t, gid_t);
int     chroot(char *);
int     cladm(int, int, void *);
int     close(int);
int     exece(const char *, const char **, const char **);
int     faccessat(int, char *, int, int);
int     fchmodat(int, char *, int, int);
int     fchownat(int, char *, uid_t, gid_t, int);
int     fcntl(int, int, intptr_t);
int64_t vfork();
int64_t forksys(int, int);
int     fstat(int, struct stat *);
int     fdsync(int, int);
int64_t getgid();
int     ucredsys(int, int, void *);
int64_t getpid();
int64_t getuid();
time_t  gtime();
int     getloadavg(int *, int);
int     rusagesys(int, void *, void *, void *, void *);
int     getpagesizes(int, size_t *, int);
int     gtty(int, intptr_t);
#if defined(__i386) || defined(__amd64)
int     hrtsys(struct hrtsysa *, rval_t *);
#endif /* __i386 || __amd64 */
int     ioctl(int, int, intptr_t);
int     kill();
int     labelsys(int, void *, void *, void *, void *, void *);
int     link(char *, char *);
int     linkat(int, char *, int, char *, int);
off32_t lseek32(int32_t, off32_t, int32_t);
off_t   lseek64(int, off_t, int);
int     lgrpsys(int, long, void *);
int     mmapobjsys(int, uint_t, mmapobj_result_t *, uint_t *, void *);
int     mknod(char *, mode_t, dev_t);
int     mknodat(int, char *, mode_t, dev_t);
int     mount(long *, rval_t *);
int     nice(int);
int     nullsys();
int     open(char *, int, int);
int     openat(int, char *, int, int);
int     pause();
long    pcsample(void *, long);
int     privsys(int, priv_op_t, priv_ptype_t, void *, size_t, int);
int     profil(unsigned short *, size_t, ulong_t, uint_t);
ssize_t pread(int, void *, size_t, off_t);
int     psecflags(procset_t *, psecflagwhich_t, secflagdelta_t *);
ssize_t pwrite(int, void *, size_t, off_t);
ssize_t read(int, void *, size_t);
int     rename(char *, char *);
int     renameat(int, char *, int, char *);
void    rexit(int);
int     semsys();
int     setgid(gid_t);
int     setpgrp(int, int, int);
int     setuid(uid_t);
uintptr_t       shmsys();
uint64_t        sidsys(int, int, int, int);
```

```
127 int     sigprocmask(int, sigset_t *, sigset_t *);
128 int     sigsuspend(sigset_t);
129 int     sigaltstack(struct sigaltstack *, struct sigaltstack *);
130 int     sigaction(int, struct sigaction *, struct sigaction *);
131 int     sigpending(int, sigset_t *);
132 int     sigresend(int, siginfo_t *, sigset_t *);
133 int     sigtimedwait(sigset_t *, siginfo_t *, timespec_t *);
134 int     getsetcontext(int, void *);
135 int     stat(char *, struct stat *);
136 int     fstatat(int, char *, struct stat *, int);
137 int     stime(time_t);
138 int     stty(int, intptr_t);
139 int     syssync();
140 int     sysacct(char *);
141 clock_t times(struct tms *);
142 long    ulimit(int, long);
143 int     getrlimit32(int, struct rlimit32 *);
144 int     setrlimit32(int, struct rlimit32 *);
145 int     umask(int);
146 int     umount2(char *, int);
147 int     unlink(char *);
148 int     unlinkat(int, char *, int);
149 int     utimesys(int, uintptr_t, uintptr_t, uintptr_t, uintptr_t);
150 int64_t utssys32(void *, int, int, void *);
151 int64_t utssys64(void *, long, int, void *);
152 int     uucopy(const void *, void *, size_t);
153 ssize_t uucopystr(const char *, char *, size_t);
154 ssize_t write(int, void *, size_t);
155 ssize_t readv(int, struct iovec *, int);
156 ssize_t writev(int, struct iovec *, int);
157 ssize_t preadv(int, struct iovec *, int, off_t, off_t);
158 ssize_t pwritev(int, struct iovec *, int, off_t, off_t);
159 int     syslwp_park(int, uintptr_t, uintptr_t);
160 int     rmdir(char *);
161 int     mkdir(char *, int);
162 int     mkdirat(int, char *, int);
163 int     getdents32(int, void *, size_t);
164 int     statfs32(char *, struct statfs32 *, int32_t, int32_t);
165 int     fstatfs32(int32_t, struct statfs32 *, int32_t, int32_t);
166 int     sysfs(int, long, long);
167 int     getmsg(int, struct strbuf *, struct strbuf *, int *);
168 int     pollsys(pollfd_t *, nfds_t, timespec_t *, sigset_t *);
169 int     putmsg(int, struct strbuf *, struct strbuf *, int);
170 int     uadmin();
171 int     lstat(char *, struct stat *);
172 int     symlink(char *, char *);
173 int     symlinkat(char *, int, char *);
174 ssize_t readlink(char *, char *, size_t);
175 ssize_t readlinkat(int, char *, char *, size_t);
176 int     resolvepath(char *, char *, size_t);
177 int     setgroups(int, gid_t *);
178 int     getgroups(int, gid_t *);
179 int     fchdir(int);
180 int     fchown(int, uid_t, uid_t);
181 int     fchmod(int, int);
182 int     getcwd(char *, size_t);
183 int     statvfs(char *, struct statvfs *);
184 int     fstatvfs(int, struct statvfs *);
185 offset_t llseek32(int32_t, uint32_t, uint32_t, int);

187 #if (defined(__i386) && !defined(__amd64)) || defined(__i386_COMPAT)
188 int     sysi86(short, uintptr_t, uintptr_t, uintptr_t);
189 #endif

191 int     acl(const char *, int, int, void *);
192 int     facl(int, int, int, void *);
```



```
193 long    priocntlsys(int, procset_t *, int, caddr_t, caddr_t);
194 int     waitsys(idtype_t, id_t, siginfo_t *, int);
195 int     sigsendsys(procset_t *, int);
196 int     mincore(caddr_t, size_t, char *);
197 caddr_t smmap64(caddr_t, size_t, int, int, int, off_t);
198 caddr_t smmap32(caddr32_t, size32_t, int, int, int, off32_t);
199 int     smmaplf32(struct mmaplf32a *, rval_t *);
200 int     mprotect(caddr_t, size_t, int);
201 int     munmap(caddr_t, size_t);
202 int     uname(struct utsname *);
203 int     lchown(char *, uid_t, gid_t);
204 int     getpmsg(int, struct strbuf *, struct strbuf *, int *, int *);
205 int     putpmsg(int, struct strbuf *, struct strbuf *, int, int);
206 int     memcntl(caddr_t, size_t, int, caddr_t, int, int);
207 long    sysconfig(int);
208 int     adjtime(struct timeval *, struct timeval *);
209 long    systeminfo(int, char *, long);
210 int     setegid(gid_t);
211 int     seteuid(uid_t);

213 int     setreuid(uid_t, uid_t);
214 int     setregid(gid_t, gid_t);
215 int     install_utrap(utrap_entry_t type, utrap_handler_t, utrap_handler_t *);
216 #ifdef __sparc
217 int     sparc_utrap_install(utrap_entry_t type, utrap_handler_t,
218     utrap_handler_t, utrap_handler_t *, utrap_handler_t *);
219 #endif

221 int     syslwp_create(ucontext_t *, int, id_t *);
222 void    syslwp_exit();
223 int     syslwp_suspend(id_t);
224 int     syslwp_continue(id_t);
225 int     syslwp_private(int, int, uintptr_t);
226 int     lwp_detach(id_t);
227 int     lwp_info(timestruc_t *);
228 int     lwp_kill(id_t, int);
229 int     lwp_self();
230 int64_t lwp_sigmask(int, uint_t, uint_t, uint_t, uint_t);
231 int     yield();
232 int     lwp_wait(id_t, id_t *);
233 int     lwp_mutex_timedlock(lwp_mutex_t *, timespec_t *, uintptr_t);
234 int     lwp_mutex_wakeup(lwp_mutex_t *, int);
235 int     lwp_mutex_unlock(lwp_mutex_t *);
236 int     lwp_mutex_trylock(lwp_mutex_t *, uintptr_t);
237 int     lwp_mutex_register(lwp_mutex_t *, caddr_t);
238 int     lwp_rwlock_sys(int, lwp_rwlock_t *, timespec_t *);
239 int     lwp_sema_post(lwp_sema_t *);
240 int     lwp_sema_timedwait(lwp_sema_t *, timespec_t *, int);
241 int     lwp_sema_trywait(lwp_sema_t *);
242 int     lwp_cond_wait(lwp_cond_t *, lwp_mutex_t *, timespec_t *, int);
243 int     lwp_cond_signal(lwp_cond_t *);
244 int     lwp_cond_broadcast(lwp_cond_t *);
245 caddr_t schedctl();

247 long    pathconf(char *, int);
248 long    fpathconf(int, int);
249 int     processor_bind(idtype_t, id_t, processorid_t, processorid_t *);
250 int     processor_info(processorid_t, processor_info_t *);
251 int     p_online(processorid_t, int);

253 /*
254  *      POSIX .4 system calls *
255  */
256 int     clock_gettime(clockid_t, timespec_t *);
257 int     clock_settime(clockid_t, timespec_t *);
258 int     clock_getres(clockid_t, timespec_t *);
```

```
259 int     timer_create(clockid_t, struct sigevent *, timer_t *);
260 int     timer_delete(timer_t);
261 int     timer_settime(timer_t, int, itimerspec_t *, itimerspec_t *);
262 int     timer_gettime(timer_t, itimerspec_t *);
263 int     timer_getoverrun(timer_t);
264 int     nanosleep(timespec_t *, timespec_t *);
265 int     sigqueue(pid_t, int, void *, int, int);
266 int     signotify(int, siginfo_t *, signotify_id_t *);

268 int     getdents64(int, void *, size_t);
269 int     stat64(char *, struct stat64 *);
270 int     lstat64(char *, struct stat64 *);
271 int     fstatat64(int, char *, struct stat64 *, int);
272 int     fstat64(int, struct stat64 *);
273 int     statvfs64(char *, struct statvfs64 *);
274 int     fstatvfs64(int, struct statvfs64 *);
275 int     setrlimit64(int, struct rlimit64 *);
276 int     getrlimit64(int, struct rlimit64 *);
277 int     pread64(int, void *, size32_t, uint32_t, uint32_t);
278 int     pwrite64(int, void *, size32_t, uint32_t, uint32_t);
279 int     open64(char *, int, int);
280 int     openat64(int, char *, int, int);

282 /*
283  * NTP syscalls
284  */

286 int ntp_gettime(struct ntptimeval *);
287 int ntp_adjtime(struct timex *);

289 /*
290  *      ++++++++++++++++++++++++
291  *      ++  SunOS4.1 Buyback  ++
292  *      ++++++++++++++++++++++++
293  *
294  *      fchroot, vhangup, gettimeofday
295  */

297 int     fchroot(int);
298 int     vhangup();
299 int     gettimeofday(struct timeval *);
300 int     getitimer(uint_t, struct itimerval *);
301 int     setitimer(uint_t, struct itimerval *, struct itimerval *);

303 int     corectl(int, uintptr_t, uintptr_t, uintptr_t);
304 int     modctl(int, uintptr_t, uintptr_t, uintptr_t, uintptr_t, uintptr_t);
305 int64_t loadable_syscall();
306 int64_t indir();

308 long    tasksys(int, projid_t, uint_t, void *, size_t);
309 long    rctlsys(int, char *, void *, void *, size_t, int);

311 long    zone();

313 int     so_socket(int, int, int, char *, int);
314 int     so_socketpair(int[2]);
315 int     bind(int, struct sockaddr *, socklen_t, int);
316 int     listen(int, int, int);
317 int     accept(int, struct sockaddr *, socklen_t *, int, int);
318 int     connect(int, struct sockaddr *, socklen_t, int);
319 int     shutdown(int, int, int);
320 ssize_t recv(int, void *, size_t, int);
321 ssize_t recvfrom(int, void *, size_t, int, struct sockaddr *, socklen_t *);
322 ssize_t recvmsg(int, struct nmsghdr *, int);
323 ssize_t send(int, void *, size_t, int);
324 ssize_t sendmsg(int, struct nmsghdr *, int);
```



```
325 ssize_t sendto(int, void *, size_t, int, struct sockaddr *, socklen_t);
326 int     getpeername(int, struct sockaddr *, socklen_t *, int);
327 int     getsockname(int, struct sockaddr *, socklen_t *, int);
328 int     getsockopt(int, int, int, void *, socklen_t *, int);
329 int     setsockopt(int, int, int, void *, socklen_t *, int);
330 int     sockconfig(int, void *, void *, void *, void *);
331 ssize_t sendfilev(int, int, const struct sendfilevec *, int, size_t *);
332 int     getrandom(void *, size_t, int);

334 typedef int64_t (*llfcn_t)();   /* for casting one-word returns */

336 /*
337  * Sysent initialization macros.
338  *      These take the name string of the system call even though that isn't
339  *      currently used in the sysent entry.  This might be useful someday.
340  *
341  * Initialization macro for system calls which take their args in the C style.
342  * These system calls return the longlong_t return value and must call
343  * set_errno() to return an error.  For SPARC, narg must be at most six.
344  * For more args, use the SYSENT_AP() routine.
345  *
346  * We are able to return two distinct values to userland via the rval_t.
347  * At this time, that corresponds to one 64-bit quantity, or two 32-bit
348  * quantities.  The kernel does not currently need to return two 64-bit
349  * values, or one 128 bit value(!), but we may do one day, so the calling
350  * sequence between userland and the kernel should permit it.
351  *
352  * The interpretation of rval_t is provided by the sy_flags field
353  * which is used to determine how to arrange the results in registers
354  * (or on the stack) for return userland.
355  */
356         /* returns a 64-bit quantity for both ABIs */
357 #define SYSENT_C(name, call, narg)      \
358         { (narg), SE_64RVAL, NULL, NULL, (llfcn_t)(call) }

360         /* returns one 32-bit value for both ABIs: r_val1 */
361 #define SYSENT_CI(name, call, narg)     \
362         { (narg), SE_32RVAL1, NULL, NULL, (llfcn_t)(call) }

364         /* returns 2 32-bit values: r_val1 & r_val2 */
365 #define SYSENT_2CI(name, call, narg)    \
366         { (narg), SE_32RVAL1|SE_32RVAL2, NULL, NULL, (llfcn_t)(call) }

368 /*
369  * Initialization macro for system calls which take their args in the standard
370  * Unix style of a pointer to the arg structure and a pointer to the rval_t.
371  *
372  * Deprecated wherever possible (slower on some architectures, and trickier
373  * to maintain two flavours).
374  */
375 #define SYSENT_AP(name, call, narg)     \
376         { (narg), SE_64RVAL, (call), NULL, syscall_ap }

378 /*
379  * Conditional constructors to build the tables without #ifdef clutter
380  */
381 #if defined(_LP64)
382 #define IF_LP64(true, false)    true
383 #else
384 #define IF_LP64(true, false)    false
385 #endif

387 #if defined(__sparc)
388 #define IF_sparc(true, false)   true
389 #else
390 #define IF_sparc(true, false)   false
```

```
391 #endif

393 #if defined(__i386) && !defined(__amd64)
394 #define IF_i386(true, false)    true
395 #else
396 #define IF_i386(true, false)    false
397 #endif

399 #if defined(__i386) || defined(__amd64)
400 #define IF_x86(true, false)     true
401 #else
402 #define IF_x86(true, false)     false
403 #endif

405 #if (defined(__i386) && !defined(__amd64)) || defined(__i386_COMPAT)
406 #define IF_386_ABI(true, false) true
407 #else
408 #define IF_386_ABI(true, false) false
409 #endif

411 /*
412  * Define system calls that return a native 'long' quantity i.e. a 32-bit
413  * or 64-bit integer - depending on how the kernel is itself compiled
414  * e.g. read(2) returns 'ssize_t' in the kernel and in userland.
415  */
416 #define SYSENT_CL(name, call, narg)     \
417         IF_LP64(SYSENT_C(name, call, narg), SYSENT_CI(name, call, narg))

419 /*
420  * Initialization macro for loadable native system calls.
421  */
422 #define SYSENT_LOADABLE()       \
423         { 0, SE_LOADABLE, (int (*)())nosys, NULL, loadable_syscall }

425 /*
426  * Initialization macro for loadable 32-bit compatibility system calls.
427  */
428 #define SYSENT_LOADABLE32()     SYSENT_LOADABLE()

430 #define SYSENT_NOSYS()          SYSENT_C("nosys", nosys, 0)

432 struct sysent nosys_ent = SYSENT_NOSYS();

434 /*
435  * Native sysent table.
436  */
437 struct sysent sysent[NSYSCALL] =
438 {
439         /*  0 */ IF_LP64(
440                         SYSENT_NOSYS(),
441                         SYSENT_C("indir",       indir,          1)),
442         /*  1 */ SYSENT_CI("exit",              rexit,          1),
443         /*  2 */ SYSENT_CI("psecflags",         psecflags,      3),
444         /*  3 */ SYSENT_CL("read",              read,           3),
445         /*  4 */ SYSENT_CL("write",             write,          3),
446         /*  5 */ SYSENT_CI("open",              open,           3),
447         /*  6 */ SYSENT_CI("close",             close,          1),
448         /*  7 */ SYSENT_CI("linkat",            linkat,         5),
449         /*  8 */ SYSENT_LOADABLE(),                     /* (was creat) */
450         /*  9 */ SYSENT_CI("link",              link,           2),
451         /* 10 */ SYSENT_CI("unlink",            unlink,         1),
452         /* 11 */ SYSENT_CI("symlinkat",         symlinkat,      3),
453         /* 12 */ SYSENT_CI("chdir",             chdir,          1),
454         /* 13 */ SYSENT_CL("time",              gtime,          0),
455         /* 14 */ SYSENT_CI("mknod",             mknod,          3),
456         /* 15 */ SYSENT_CI("chmod",             chmod,          2),
```



```
457         /* 16 */ SYSENT_CI("chown",             chown,          3),
458         /* 17 */ SYSENT_CI("brk",               brk,            1),
459         /* 18 */ SYSENT_CI("stat",              stat,           2),
460         /* 19 */ IF_LP64(
461                         SYSENT_CL("lseek",      lseek64,        3),
462                         SYSENT_CL("lseek",      lseek32,        3)),
463         /* 20 */ SYSENT_2CI("getpid",           getpid,         0),
464         /* 21 */ SYSENT_AP("mount",             mount,          8),
465         /* 22 */ SYSENT_CL("readlinkat",        readlinkat,     4),
466         /* 23 */ SYSENT_CI("setuid",            setuid,         1),
467         /* 24 */ SYSENT_2CI("getuid",           getuid,         0),
468         /* 25 */ SYSENT_CI("stime",             stime,          1),
469         /* 26 */ SYSENT_CL("pcsample",          pcsample,       2),
470         /* 27 */ SYSENT_CI("alarm",             alarm,          1),
471         /* 28 */ SYSENT_CI("fstat",             fstat,          2),
472         /* 29 */ SYSENT_CI("pause",             pause,          0),
473         /* 30 */ SYSENT_LOADABLE(),                    /* (was utime) */
474         /* 31 */ SYSENT_CI("stty",              stty,           2),
475         /* 32 */ SYSENT_CI("gtty",              gtty,           2),
476         /* 33 */ SYSENT_CI("access",            access,         2),
477         /* 34 */ SYSENT_CI("nice",              nice,           1),
478         /* 35 */ IF_LP64(
479                         SYSENT_NOSYS(),
480                         SYSENT_CI("statfs",     statfs32,       4)),
481         /* 36 */ SYSENT_CI("sync",              syssync,        0),
482         /* 37 */ SYSENT_CI("kill",              kill,           2),
483         /* 38 */ IF_LP64(
484                         SYSENT_NOSYS(),
485                         SYSENT_CI("fstatfs",    fstatfs32,      4)),
486         /* 39 */ SYSENT_CI("setpgrp",           setpgrp,        3),
487         /* 40 */ SYSENT_CI("uucopystr",         uucopystr,      3),
488         /* 41 */ SYSENT_LOADABLE(),                    /* (was dup) */
489         /* 42 */ SYSENT_LOADABLE(),                    /* pipe */
490         /* 43 */ SYSENT_CL("times",             times,          1),
491         /* 44 */ SYSENT_CI("prof",              profil,         4),
492         /* 45 */ SYSENT_CI("faccessat",         faccessat,      4),
493         /* 46 */ SYSENT_CI("setgid",            setgid,         1),
494         /* 47 */ SYSENT_2CI("getgid",           getgid,         0),
495         /* 48 */ SYSENT_CI("mknodat",           mknodat,        4),
496         /* 49 */ SYSENT_LOADABLE(),                    /* msgsys */
497         /* 50 */ IF_x86(
498                         SYSENT_CI("sysi86",     sysi86,         4),
499                         SYSENT_LOADABLE()),             /* (was sys3b) */
500         /* 51 */ SYSENT_LOADABLE(),                    /* sysacct */
501         /* 52 */ SYSENT_LOADABLE(),                    /* shmsys */
502         /* 53 */ SYSENT_LOADABLE(),                    /* semsys */
503         /* 54 */ SYSENT_CI("ioctl",             ioctl,          3),
504         /* 55 */ SYSENT_CI("uadmin",            uadmin,         3),
505         /* 56 */ SYSENT_CI("fchownat",          fchownat,       5),
506         /* 57 */ IF_LP64(
507                         SYSENT_2CI("utssys",    utssys64,       4),
508                         SYSENT_2CI("utssys",    utssys32,       4)),
509         /* 58 */ SYSENT_CI("fdsync",            fdsync,         2),
510         /* 59 */ SYSENT_CI("exece",             exece,          3),
511         /* 60 */ SYSENT_CI("umask",             umask,          1),
512         /* 61 */ SYSENT_CI("chroot",            chroot,         1),
513         /* 62 */ SYSENT_CI("fcntl",             fcntl,          3),
514         /* 63 */ SYSENT_CI("ulimit",            ulimit,         2),
515         /* 64 */ SYSENT_CI("renameat",          renameat,       4),
516         /* 65 */ SYSENT_CI("unlinkat",          unlinkat,       3),
517         /* 66 */ SYSENT_CI("fstatat",           fstatat,        4),
518         /* 67 */ IF_LP64(
519                         SYSENT_NOSYS(),
520                         SYSENT_CI("fstatat64",  fstatat64,      4)),
521         /* 68 */ SYSENT_CI("openat",            openat,         4),
522         /* 69 */ IF_LP64(
```

```
523                     SYSENT_NOSYS(),
524                     SYSENT_CI("openat64",   openat64,       4)),
525         /* 70 */ SYSENT_CI("tasksys",          tasksys,        5),
526         /* 71 */ SYSENT_LOADABLE(),            /* acctctl */
527         /* 72 */ SYSENT_LOADABLE(),            /* exacct */
528         /* 73 */ SYSENT_CI("getpagesizes",     getpagesizes,   3),
529         /* 74 */ SYSENT_CI("rctlsys",          rctlsys,        6),
530         /* 75 */ SYSENT_2CI("sidsys",          sidsys,         4),
531         /* 76 */ SYSENT_LOADABLE(),                        /* (was fsat) */
532         /* 77 */ SYSENT_CI("lwp_park",         syslwp_park,    3),
533         /* 78 */ SYSENT_CL("sendfilev",        sendfilev,      5),
534         /* 79 */ SYSENT_CI("rmdir",            rmdir,          1),
535         /* 80 */ SYSENT_CI("mkdir",            mkdir,          2),
536         /* 81 */ IF_LP64(
537                     SYSENT_CI("getdents",   getdents64,     3),
538                     SYSENT_CI("getdents",   getdents32,     3)),
539         /* 82 */ SYSENT_CI("privsys",          privsys,        6),
540         /* 83 */ SYSENT_CI("ucredsys",         ucredsys,       3),
541         /* 84 */ SYSENT_CI("sysfs",            sysfs,          3),
542         /* 85 */ SYSENT_CI("getmsg",           getmsg,         4),
543         /* 86 */ SYSENT_CI("putmsg",           putmsg,         4),
544         /* 87 */ SYSENT_LOADABLE(),                        /* (was poll) */
545         /* 88 */ SYSENT_CI("lstat",            lstat,          2),
546         /* 89 */ SYSENT_CI("symlink",          symlink,        2),
547         /* 90 */ SYSENT_CL("readlink",         readlink,       3),
548         /* 91 */ SYSENT_CI("setgroups",        setgroups,      2),
549         /* 92 */ SYSENT_CI("getgroups",        getgroups,      2),
550         /* 93 */ SYSENT_CI("fchmod",           fchmod,         2),
551         /* 94 */ SYSENT_CI("fchown",           fchown,         3),
552         /* 95 */ SYSENT_CI("sigprocmask",      sigprocmask,    3),
553         /* 96 */ SYSENT_CI("sigsuspend",       sigsuspend,     1),
554         /* 97 */ SYSENT_CI("sigaltstack",      sigaltstack,    2),
555         /* 98 */ SYSENT_CI("sigaction",        sigaction,      3),
556         /* 99 */ SYSENT_CI("sigpending",       sigpending,     2),
557         /* 100 */ SYSENT_CI("getsetcontext",   getsetcontext,  2),
558         /* 101 */ SYSENT_CI("fchmodat",        fchmodat,       4),
559         /* 102 */ SYSENT_CI("mkdirat",         mkdirat,        3),
560         /* 103 */ SYSENT_CI("statvfs",         statvfs,        2),
561         /* 104 */ SYSENT_CI("fstatvfs",        fstatvfs,       2),
562         /* 105 */ SYSENT_CI("getloadavg",      getloadavg,     2),
563         /* 106 */ SYSENT_LOADABLE(),           /* nfssys */
564         /* 107 */ SYSENT_CI("waitsys",         waitsys,        4),
565         /* 108 */ SYSENT_CI("sigsendset",      sigsendsys,     2),
566         /* 109 */ IF_x86(
567                     SYSENT_AP("hrtsys",     hrtsys,         5),
568                     SYSENT_LOADABLE()),
569         /* 110 */ SYSENT_CI("utimesys",        utimesys,       5),
570         /* 111 */ SYSENT_CI("sigresend",       sigresend,      3),
571         /* 112 */ SYSENT_CL("priocntlsys",     priocntlsys,    5),
572         /* 113 */ SYSENT_CL("pathconf",        pathconf,       2),
573         /* 114 */ SYSENT_CI("mincore",         mincore,        3),
574         /* 115 */ IF_LP64(
575                     SYSENT_CL("mmap",       smmap64,        6),
576                     SYSENT_CL("mmap",       smmap32,        6)),
577         /* 116 */ SYSENT_CI("mprotect",        mprotect,       3),
578         /* 117 */ SYSENT_CI("munmap",          munmap,         2),
579         /* 118 */ SYSENT_CL("fpathconf",       fpathconf,      2),
580         /* 119 */ SYSENT_2CI("vfork",          vfork,          0),
581         /* 120 */ SYSENT_CI("fchdir",          fchdir,         1),
582         /* 121 */ SYSENT_CL("readv",           readv,          3),
583         /* 122 */ SYSENT_CL("writev",          writev,         3),
584         /* 123 */ SYSENT_CL("preadv",          preadv,         5),
585         /* 124 */ SYSENT_CL("pwritev",         pwritev,        5),
586         /* 125 */ SYSENT_LOADABLE(),                       /* (was fxstat) */
587         /* 126 */ SYSENT_CI("getrandom",       getrandom,      3),
588         /* 127 */ SYSENT_CI("mmapobj",         mmapobjsys,     5),
```



```
589         /* 128 */ IF_LP64(
590                     SYSENT_CI("setrlimit",  setrlimit64,    2),
591                     SYSENT_CI("setrlimit",  setrlimit32,    2)),
592         /* 129 */ IF_LP64(
593                     SYSENT_CI("getrlimit",  getrlimit64,    2),
594                     SYSENT_CI("getrlimit",  getrlimit32,    2)),
595         /* 130 */ SYSENT_CI("lchown",          lchown,         3),
596         /* 131 */ SYSENT_CI("memcntl",         memcntl,        6),
597         /* 132 */ SYSENT_CI("getpmsg",         getpmsg,        5),
598         /* 133 */ SYSENT_CI("putpmsg",         putpmsg,        5),
599         /* 134 */ SYSENT_CI("rename",          rename,         2),
600         /* 135 */ SYSENT_CI("uname",           uname,          1),
601         /* 136 */ SYSENT_CI("setegid",         setegid,        1),
602         /* 137 */ SYSENT_CL("sysconfig",       sysconfig,      1),
603         /* 138 */ SYSENT_CI("adjtime",         adjtime,        2),
604         /* 139 */ SYSENT_CL("systeminfo",      systeminfo,     3),
605         /* 140 */ SYSENT_LOADABLE(),           /* sharefs */
606         /* 141 */ SYSENT_CI("seteuid",         seteuid,        1),
607         /* 142 */ SYSENT_2CI("forksys",        forksys,        2),
608         /* 143 */ SYSENT_LOADABLE(),                   /* (was fork1) */
609         /* 144 */ SYSENT_CI("sigtimedwait",    sigtimedwait,   3),
610         /* 145 */ SYSENT_CI("lwp_info",        lwp_info,       1),
611         /* 146 */ SYSENT_CI("yield",           yield,          0),
612         /* 147 */ SYSENT_LOADABLE(),               /* (was lwp_sema_wait) */
613         /* 148 */ SYSENT_CI("lwp_sema_post",   lwp_sema_post,  1),
614         /* 149 */ SYSENT_CI("lwp_sema_trywait", lwp_sema_trywait, 1),
615         /* 150 */ SYSENT_CI("lwp_detach",      lwp_detach,     1),
616         /* 151 */ SYSENT_CI("corectl",         corectl,        4),
617         /* 152 */ SYSENT_CI("modctl",          modctl,         6),
618         /* 153 */ SYSENT_CI("fchroot",         fchroot,        1),
619         /* 154 */ SYSENT_LOADABLE(),                   /* (was utimes) */
620         /* 155 */ SYSENT_CI("vhangup",         vhangup,        0),
621         /* 156 */ SYSENT_CI("gettimeofday",    gettimeofday,   1),
622         /* 157 */ SYSENT_CI("getitimer",       getitimer,      2),
623         /* 158 */ SYSENT_CI("setitimer",       setitimer,      3),
624         /* 159 */ SYSENT_CI("lwp_create",      syslwp_create,  3),
625         /* 160 */ SYSENT_CI("lwp_exit", (int (*)())syslwp_exit, 0),
626         /* 161 */ SYSENT_CI("lwp_suspend",     syslwp_suspend, 1),
627         /* 162 */ SYSENT_CI("lwp_continue",    syslwp_continue, 1),
628         /* 163 */ SYSENT_CI("lwp_kill",        lwp_kill,       2),
629         /* 164 */ SYSENT_CI("lwp_self",        lwp_self,       0),
630         /* 165 */ SYSENT_2CI("lwp_sigmask",    lwp_sigmask,    5),
631         /* 166 */ IF_x86(
632                     SYSENT_CI("lwp_private", syslwp_private, 3),
633                     SYSENT_NOSYS()),
634         /* 167 */ SYSENT_CI("lwp_wait",        lwp_wait,       2),
635         /* 168 */ SYSENT_CI("lwp_mutex_wakeup", lwp_mutex_wakeup,      2),
636         /* 169 */ SYSENT_LOADABLE(),               /* (was lwp_mutex_lock) */
637         /* 170 */ SYSENT_CI("lwp_cond_wait",   lwp_cond_wait,          4),
638         /* 171 */ SYSENT_CI("lwp_cond_signal", lwp_cond_signal,        1),
639         /* 172 */ SYSENT_CI("lwp_cond_broadcast", lwp_cond_broadcast,  1),
640         /* 173 */ SYSENT_CL("pread",           pread,          4),
641         /* 174 */ SYSENT_CL("pwrite ",         pwrite,         4),
642         /*
643          * The 64-bit C library maps llseek() to lseek(), so this
644          * is needed as a native syscall only on the 32-bit kernel.
645          */
646         /* 175 */ IF_LP64(
647                     SYSENT_NOSYS(),
648                     SYSENT_C("llseek",      llseek32,       4)),
649         /* 176 */ SYSENT_LOADABLE(),           /* inst_sync */
650         /* 177 */ SYSENT_CI("brandsys",        brandsys,       6),
651         /* 178 */ SYSENT_LOADABLE(),           /* kaio */
652         /* 179 */ SYSENT_LOADABLE(),           /* cpc */
653         /* 180 */ SYSENT_CI("lgrpsys",         lgrpsys,        3),
654         /* 181 */ SYSENT_CI("rusagesys",       rusagesys,      5),
```

```
655         /* 182 */ SYSENT_LOADABLE(),            /* portfs */
656         /* 183 */ SYSENT_CI("pollsys",          pollsys,        4),
657         /* 184 */ SYSENT_CI("labelsys",         labelsys,       5),
658         /* 185 */ SYSENT_CI("acl",              acl,            4),
659         /* 186 */ SYSENT_AP("auditsys",         auditsys,       6),
660         /* 187 */ SYSENT_CI("processor_bind",   processor_bind, 4),
661         /* 188 */ SYSENT_CI("processor_info",   processor_info, 2),
662         /* 189 */ SYSENT_CI("p_online",         p_online,       2),
663         /* 190 */ SYSENT_CI("sigqueue",         sigqueue,       5),
664         /* 191 */ SYSENT_CI("clock_gettime",    clock_gettime,  2),
665         /* 192 */ SYSENT_CI("clock_settime",    clock_settime,  2),
666         /* 193 */ SYSENT_CI("clock_getres",     clock_getres,   2),
667         /* 194 */ SYSENT_CI("timer_create",     timer_create,   3),
668         /* 195 */ SYSENT_CI("timer_delete",     timer_delete,   1),
669         /* 196 */ SYSENT_CI("timer_settime",    timer_settime,  4),
670         /* 197 */ SYSENT_CI("timer_gettime",    timer_gettime,  2),
671         /* 198 */ SYSENT_CI("timer_getoverrun", timer_getoverrun, 1),
672         /* 199 */ SYSENT_CI("nanosleep",        nanosleep,      2),
673         /* 200 */ SYSENT_CI("facl",             facl,           4),
674         /* 201 */ SYSENT_LOADABLE(),            /* door */
675         /* 202 */ SYSENT_CI("setreuid",         setreuid,       2),
676         /* 203 */ SYSENT_CI("setregid",         setregid,       2),
677         /* 204 */ SYSENT_CI("install_utrap",    install_utrap,  3),
678         /* 205 */ SYSENT_CI("signotify",        signotify,      3),
679         /* 206 */ SYSENT_CL("schedctl",         schedctl,       0),
680         /* 207 */ SYSENT_LOADABLE(),            /* pset */
681         /* 208 */ IF_sparc(
682                    SYSENT_CI("sparc_utrap_install", sparc_utrap_install, 5),
683                    SYSENT_NOSYS()),
684         /* 209 */ SYSENT_CI("resolvepath",      resolvepath,    3),
685         /* 210 */ SYSENT_CI("lwp_mutex_timedlock", lwp_mutex_timedlock, 3),
686         /* 211 */ SYSENT_CI("lwp_sema_timedwait", lwp_sema_timedwait,   3),
687         /* 212 */ SYSENT_CI("lwp_rwlock_sys",   lwp_rwlock_sys, 3),
688         /*
689          * Syscalls 213-225: 32-bit system call support for large files.
690          *
691          * (The 64-bit C library transparently maps these system calls
692          * back to their native versions, so almost all of them are only
693          * needed as native syscalls on the 32-bit kernel).
694          */
695         /* 213 */ IF_LP64(
696                       SYSENT_NOSYS(),
697                       SYSENT_CI("getdents64", getdents64,     3)),
698         /* 214 */ IF_LP64(
699                       SYSENT_NOSYS(),
700                       SYSENT_AP("smmaplf32",  smmaplf32,      7)),
701         /* 215 */ IF_LP64(
702                       SYSENT_NOSYS(),
703                       SYSENT_CI("stat64",     stat64,         2)),
704         /* 216 */ IF_LP64(
705                       SYSENT_NOSYS(),
706                       SYSENT_CI("lstat64",    lstat64,        2)),
707         /* 217 */ IF_LP64(
708                       SYSENT_NOSYS(),
709                       SYSENT_CI("fstat64",    fstat64,        2)),
710         /* 218 */ IF_LP64(
711                       SYSENT_NOSYS(),
712                       SYSENT_CI("statvfs64",  statvfs64,      2)),
713         /* 219 */ IF_LP64(
714                       SYSENT_NOSYS(),
715                       SYSENT_CI("fstatvfs64", fstatvfs64,     2)),
716         /* 220 */ IF_LP64(
717                       SYSENT_NOSYS(),
718                       SYSENT_CI("setrlimit64", setrlimit64,   2)),
719         /* 221 */ IF_LP64(
720                       SYSENT_NOSYS(),
```



```
721                       SYSENT_CI("getrlimit64", getrlimit64,   2)),
722         /* 222 */ IF_LP64(
723                       SYSENT_NOSYS(),
724                       SYSENT_CI("pread64",    pread64,        5)),
725         /* 223 */ IF_LP64(
726                       SYSENT_NOSYS(),
727                       SYSENT_CI("pwrite64",   pwrite64,       5)),
728         /* 224 */ SYSENT_LOADABLE(),                   /* (was creat64) */
729         /* 225 */ IF_LP64(
730                       SYSENT_NOSYS(),
731                       SYSENT_CI("open64",     open64,         3)),
732         /* 226 */ SYSENT_LOADABLE(),            /* rpcsys */
733         /* 227 */ SYSENT_CL("zone",             zone,           5),
734         /* 228 */ SYSENT_LOADABLE(),            /* autofssys */
735         /* 229 */ SYSENT_CI("getcwd",           getcwd,         2),
736         /* 230 */ SYSENT_CI("so_socket",        so_socket,      5),
737         /* 231 */ SYSENT_CI("so_socketpair",    so_socketpair,  1),
738         /* 232 */ SYSENT_CI("bind",             bind,           4),
739         /* 233 */ SYSENT_CI("listen",           listen,         3),
740         /* 234 */ SYSENT_CI("accept",           accept,         5),
741         /* 235 */ SYSENT_CI("connect",          connect,        4),
742         /* 236 */ SYSENT_CI("shutdown",         shutdown,       3),
743         /* 237 */ SYSENT_CL("recv",             recv,           4),
744         /* 238 */ SYSENT_CL("recvfrom",         recvfrom,       6),
745         /* 239 */ SYSENT_CL("recvmsg",          recvmsg,        3),
746         /* 240 */ SYSENT_CL("send",             send,           4),
747         /* 241 */ SYSENT_CL("sendmsg",          sendmsg,        3),
748         /* 242 */ SYSENT_CL("sendto",           sendto,         6),
749         /* 243 */ SYSENT_CI("getpeername",      getpeername,    4),
750         /* 244 */ SYSENT_CI("getsockname",      getsockname,    4),
751         /* 245 */ SYSENT_CI("getsockopt",       getsockopt,     6),
752         /* 246 */ SYSENT_CI("setsockopt",       setsockopt,     6),
753         /* 247 */ SYSENT_CI("sockconfig",       sockconfig,     5),
754         /* 248 */ SYSENT_CI("ntp_gettime",      ntp_gettime,    1),
755         /* 249 */ SYSENT_CI("ntp_adjtime",      ntp_adjtime,    1),
756         /* 250 */ SYSENT_CI("lwp_mutex_unlock", lwp_mutex_unlock,       1),
757         /* 251 */ SYSENT_CI("lwp_mutex_trylock", lwp_mutex_trylock,     2),
758         /* 252 */ SYSENT_CI("lwp_mutex_register", lwp_mutex_register,   2),
759         /* 253 */ SYSENT_CI("cladm",            cladm,          3),
760         /* 254 */ SYSENT_CI("uucopy",           uucopy,         3),
761         /* 255 */ SYSENT_CI("umount2",          umount2,        2)
762 };
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   11477 Thu Jun 30 21:58:45 2016
new/usr/src/uts/sun4u/vm/mach_vm_dep.c
Code review comments from pmooney (sundry), and igork (screwups in zonecfg refac
**********************************************************
______unchanged_portion_omitted_

 140 /*
 141  * The maximum amount a randomized mapping will be slewed.  We should perhaps
 142  * arrange things so these tunables can be separate for mmap, mmapobj, and
 143  * ld.so
 144  */
 145 size_t aslr_max_map_skew = 256 * 1024 * 1024; /* 256MB */

 147 /*
 148  * map_addr_proc() is the routine called when the system is to
 149  * choose an address for the user.  We will pick an address
 150  * range which is just below the current stack limit.  The
 151  * algorithm used for cache consistency on machines with virtual
 152  * address caches is such that offset 0 in the vnode is always
 153  * on a shm_alignment'ed aligned address.  Unfortunately, this
 154  * means that vnodes which are demand paged will not be mapped
 155  * cache consistently with the executable images.  When the
 156  * cache alignment for a given object is inconsistent, the
 157  * lower level code must manage the translations so that this
 158  * is not seen here (at the cost of efficiency, of course).
 159  *
 160  * Every mapping will have a redzone of a single page on either side of
 161  * the request. This is done to leave one page unmapped between segments.
 162  * This is not required, but it's useful for the user because if their
 163  * program strays across a segment boundary, it will catch a fault
 164  * immediately making debugging a little easier.  Currently the redzone
 165  * is mandatory.
 166  *
 167  *
 168  * addrp is a value/result parameter.
 169  *      On input it is a hint from the user to be used in a completely
 170  *      machine dependent fashion.  For MAP_ALIGN, addrp contains the
 171  *      minimal alignment, which must be some "power of two" multiple of
 172  *      pagesize.
 173  *
 174  *      On output it is NULL if no address can be found in the current
 175  *      processes address space or else an address that is currently
 176  *      not mapped for len bytes with a page of red zone on either side.
 177  *      If vacalign is true, then the selected address will obey the alignment
 178  *      constraints of a vac machine based on the given off value.
 179  */
 180 /*ARGSUSED4*/
 181 void
 182 map_addr_proc(caddr_t *addrp, size_t len, offset_t off, int vacalign,
 183     caddr_t userlimit, struct proc *p, uint_t flags)
 184 {
 185         struct as *as = p->p_as;
 186         caddr_t addr;
 187         caddr_t base;
 188         size_t slen;
 189         uintptr_t align_amount;
 190         int allow_largepage_alignment = 1;

 192         base = p->p_brkbase;
 193         if (userlimit < as->a_userlimit) {
 194                 /*
 195                  * This happens when a program wants to map something in
 196                  * a range that's accessible to a program in a smaller
 197                  * address space.  For example, a 64-bit program might
 198                  * be calling mmap32(2) to guarantee that the returned
```



```
 199                  * address is below 4Gbytes.
 200                  */
 201                 ASSERT(userlimit > base);
 202                 slen = userlimit - base;
 203         } else {
 204                 slen = p->p_usrstack - base -
 205                     ((p->p_stk_ctl + PAGEOFFSET) & PAGEMASK);
 206         }

 208         /* Make len be a multiple of PAGESIZE */
 209         len = (len + PAGEOFFSET) & PAGEMASK;

 211         /*
 212          *  If the request is larger than the size of a particular
 213          *  mmu level, then we use that level to map the request.
 214          *  But this requires that both the virtual and the physical
 215          *  addresses be aligned with respect to that level, so we
 216          *  do the virtual bit of nastiness here.
 217          *
 218          *  For 32-bit processes, only those which have specified
 219          *  MAP_ALIGN or an addr will be aligned on a page size > 4MB. Otherwise
 220          *  we can potentially waste up to 256MB of the 4G process address
 221          *  space just for alignment.
 222          */
 223         if (p->p_model == DATAMODEL_ILP32 && ((flags & MAP_ALIGN) == 0 ||
 224             ((uintptr_t)*addrp) != 0)) {
 225                 allow_largepage_alignment = 0;
 226         }
 227         if ((mmu_page_sizes == max_mmu_page_sizes) &&
 228             allow_largepage_alignment &&
 229             (len >= MMU_PAGESIZE256M)) {        /* 256MB mappings */
 230                 align_amount = MMU_PAGESIZE256M;
 231         } else if ((mmu_page_sizes == max_mmu_page_sizes) &&
 232             allow_largepage_alignment &&
 233             (len >= MMU_PAGESIZE32M)) { /* 32MB mappings */
 234                 align_amount = MMU_PAGESIZE32M;
 235         } else if (len >= MMU_PAGESIZE4M) {  /* 4MB mappings */
 236                 align_amount = MMU_PAGESIZE4M;
 237         } else if (len >= MMU_PAGESIZE512K) { /* 512KB mappings */
 238                 align_amount = MMU_PAGESIZE512K;
 239         } else if (len >= MMU_PAGESIZE64K) { /* 64KB mappings */
 240                 align_amount = MMU_PAGESIZE64K;
 241         } else  {
 242                 /*
 243                  * Align virtual addresses on a 64K boundary to ensure
 244                  * that ELF shared libraries are mapped with the appropriate
 245                  * alignment constraints by the run-time linker.
 246                  */
 247                 align_amount = ELF_SPARC_MAXPGSZ;
 248                 if ((flags & MAP_ALIGN) && ((uintptr_t)*addrp != 0) &&
 249                     ((uintptr_t)*addrp < align_amount))
 250                         align_amount = (uintptr_t)*addrp;
 251         }

 253         /*
 254          * 64-bit processes require 1024K alignment of ELF shared libraries.
 255          */
 256         if (p->p_model == DATAMODEL_LP64)
 257                 align_amount = MAX(align_amount, ELF_SPARCV9_MAXPGSZ);
 258 #ifdef VAC
 259         if (vac && vacalign && (align_amount < shm_alignment))
 260                 align_amount = shm_alignment;
 261 #endif

 263         if ((flags & MAP_ALIGN) && ((uintptr_t)*addrp > align_amount)) {
 264                 align_amount = (uintptr_t)*addrp;
```

```
265         }

267         ASSERT(ISP2(align_amount));
268         ASSERT(align_amount == 0 || align_amount >= PAGESIZE);

270         /*
271          * Look for a large enough hole starting below the stack limit.
272          * After finding it, use the upper part.
273          */
274         as_purge(as);
275         off = off & (align_amount - 1);

277         if (as_gap_aligned(as, len, &base, &slen, AH_HI, NULL, align_amount,
278             PAGESIZE, off) == 0) {
279                 caddr_t as_addr;

281                 /*
282                  * addr is the highest possible address to use since we have
283                  * a PAGESIZE redzone at the beginning and end.
284                  */
285                 addr = base + slen - (PAGESIZE + len);
286                 as_addr = addr;
287                 /*
288                  * Round address DOWN to the alignment amount and
289                  * add the offset in.
290                  * If addr is greater than as_addr, len would not be large
291                  * enough to include the redzone, so we must adjust down
292                  * by the alignment amount.
293                  */
294                 addr = (caddr_t)((uintptr_t)addr & (~(align_amount - 1l)));
295                 addr += (long)off;
296                 if (addr > as_addr) {
297                         addr -= align_amount;
298                 }

300                 /*
301                  * If randomization is requested, slew the allocation
302                  * backwards, within the same gap, by a random amount.
303                  */
304                 if (flags & _MAP_RANDOMIZE) {
305                         uint32_t slew;
306                         uint32_t maxslew;

308                         (void) random_get_pseudo_bytes((uint8_t *)&slew,
309                             sizeof (slew));

311                         maxslew = MIN(aslr_max_map_skew, (addr - base));
312                         /*
313                          * Don't allow ASLR to cause mappings to fail below
314                          * because of SF erratum #57
315                          */
316                         maxslew = MIN(maxslew, (addr - errata57_limit));

318                         slew = slew % maxslew;
318                         slew = slew % MIN(MIN(aslr_max_map_skew, (addr - base)),
319                             addr - errata57_limit);
319                         addr -= P2ALIGN(slew, align_amount);
320                 }

322                 ASSERT(addr > base);
323                 ASSERT(addr + len < base + slen);
324                 ASSERT(((uintptr_t)addr & (align_amount - 1l)) ==
325                     ((uintptr_t)(off)));
326                 *addrp = addr;

328 #if defined(SF_ERRATA_57)
```



```
329                 if (AS_TYPE_64BIT(as) && addr < errata57_limit) {
330                         *addrp = NULL;
331                 }
332 #endif
333         } else {
334                 *addrp = NULL;  /* no more virtual space */
335         }
336 }
```

_____***unchanged_portion_omitted_***