```
**********************************************************
   38217 Sun Mar  1 12:58:18 2015
new/usr/src/cmd/ptools/pargs/pargs.c
5558 pargs is missing -l option in usage
Reviewed by: Marcel Telka <marcel@telka.sk>
Reviewed by: Josef 'Jeff' Sipek <josef.sipek@nexenta.com>
**********************************************************
______unchanged_portion_omitted_

1268 int
1269 main(int argc, char *argv[])
1270 {
1271         int aflag = 0, cflag = 0, eflag = 0, xflag = 0, lflag = 0;
1272         int errflg = 0, retc = 0;
1273         int opt;
1274         int error = 1;
1275         core_content_t content = 0;

1277         (void) setlocale(LC_ALL, "");

1279         if ((command = strrchr(argv[0], '/')) != NULL)
1280                 command++;
1281         else
1282                 command = argv[0];

1284         while ((opt = getopt(argc, argv, "acelxF")) != EOF) {
1285                 switch (opt) {
1286                 case 'a':               /* show process arguments */
1287                         content |= CC_CONTENT_STACK;
1288                         aflag++;
1289                         break;
1290                 case 'c':               /* force 7-bit ascii */
1291                         cflag++;
1292                         break;
1293                 case 'e':               /* show environment variables */
1294                         content |= CC_CONTENT_STACK;
1295                         eflag++;
1296                         break;
1297                 case 'l':
1298                         lflag++;
1299                         aflag++;        /* -l implies -a */
1300                         break;
1301                 case 'x':               /* show aux vector entries */
1302                         xflag++;
1303                         break;
1304                 case 'F':
1305                         /*
1306                          * Since we open the process read-only, there is no need
1307                          * for the -F flag.  It's a documented flag, so we
1308                          * consume it silently.
1309                          */
1310                         break;
1311                 default:
1312                         errflg++;
1313                         break;
1314                 }
1315         }

1317         /* -a is the default if no options are specified */
1318         if ((aflag + eflag + xflag + lflag) == 0) {
1319                 aflag++;
1320                 content |= CC_CONTENT_STACK;
1321         }

1323         /* -l cannot be used with the -x or -e flags */
1324         if (lflag && (xflag || eflag)) {
```



```
1325                 (void) fprintf(stderr, "-l is incompatible with -x and -e\n");
1326                 errflg++;
1327         }

1329         argc -= optind;
1330         argv += optind;

1332         if (errflg || argc <= 0) {
1333                 (void) fprintf(stderr,
1334                     "usage:  %s [-aceFlx] { pid | core } ...\n"
1334                     "usage:  %s [-acexF] { pid | core } ...\n"
1335                     "  (show process arguments and environment)\n"
1336                     "  -a: show process arguments (default)\n"
1337                     "  -c: interpret characters as 7-bit ascii regardless of "
1338                     "locale\n"
1339                     "  -e: show environment variables\n"
1340                     "  -F: force grabbing of the target process\n"
1341 #endif /* ! codereview */
1342                     "  -l: display arguments as command line\n"
1343                     "  -x: show aux vector entries\n", command);
1340                     "  -x: show aux vector entries\n"
1341                     "  -F: force grabbing of the target process\n", command);
1344                 return (2);
1345         }

1347         while (argc-- > 0) {
1348                 char *arg;
1349                 int gret, r;
1350                 psinfo_t psinfo;
1351                 char *psargs_conv;
1352                 struct ps_prochandle *Pr;
1353                 pargs_data_t datap;
1354                 char *info;
1355                 size_t info_sz;
1356                 int pstate;
1357                 char execname[PATH_MAX];
1358                 int unprintable;
1359                 int diflocale;

1361                 (void) fflush(stdout);
1362                 arg = *argv++;

1364                 /*
1365                  * Suppress extra blanks lines if we've encountered processes
1366                  * which can't be opened.
1367                  */
1368                 if (error == 0) {
1369                         (void) printf("\n");
1370                 }
1371                 error = 0;

1373                 /*
1374                  * First grab just the psinfo information, in case this
1375                  * process is a zombie (in which case proc_arg_grab() will
1376                  * fail).  If so, print a nice message and continue.
1377                  */
1378                 if (proc_arg_psinfo(arg, PR_ARG_ANY, &psinfo,
1379                     &gret) == -1) {
1380                         (void) fprintf(stderr, "%s: cannot examine %s: %s\n",
1381                             command, arg, Pgrab_error(gret));
1382                         retc++;
1383                         error = 1;
1384                         continue;
1385                 }

1387                 if (psinfo.pr_nlwp == 0) {
```

```
1388                        (void) printf("%d: <defunct>\n", (int)psinfo.pr_pid);
1389                        continue;
1390                }

1392                /*
1393                 * If process is a "system" process (like pageout), just
1394                 * print its psargs and continue on.
1395                 */
1396                if (psinfo.pr_size == 0 && psinfo.pr_rssize == 0) {
1397                        proc_unctrl_psinfo(&psinfo);
1398                        if (!lflag)
1399                                (void) printf("%d: ", (int)psinfo.pr_pid);
1400                        (void) printf("%s\n", psinfo.pr_psargs);
1401                        continue;
1402                }

1404                /*
1405                 * Open the process readonly, since we do not need to write to
1406                 * the control file.
1407                 */
1408                if ((Pr = proc_arg_grab(arg, PR_ARG_ANY, PGRAB_RDONLY,
1409                    &gret)) == NULL) {
1410                        (void) fprintf(stderr, "%s: cannot examine %s: %s\n",
1411                            command, arg, Pgrab_error(gret));
1412                        retc++;
1413                        error = 1;
1414                        continue;
1415                }

1417                pstate = Pstate(Pr);

1419                if (pstate == PS_DEAD &&
1420                    (Pcontent(Pr) & content) != content) {
1421                        (void) fprintf(stderr, "%s: core '%s' has "
1422                            "insufficient content\n", command, arg);
1423                        retc++;
1424                        continue;
1425                }

1427                /*
1428                 * If malloc() fails, we return here so that we can let go
1429                 * of the victim, restore our locale, print a message,
1430                 * then exit.
1431                 */
1432                if ((r = setjmp(env)) != 0) {
1433                        Prelease(Pr, 0);
1434                        (void) setlocale(LC_ALL, "");
1435                        (void) fprintf(stderr, "%s: out of memory: %s\n",
1436                            command, strerror(r));
1437                        return (1);
1438                }

1440                dmodel = Pstatus(Pr)->pr_dmodel;
1441                bzero(&datap, sizeof (datap));
1442                bcopy(Ppsinfo(Pr), &psinfo, sizeof (psinfo_t));
1443                datap.pd_proc = Pr;
1444                datap.pd_psinfo = &psinfo;

1446                if (cflag)
1447                        datap.pd_conv_flags |= CONV_STRICT_ASCII;

1449                /*
1450                 * Strip control characters, then record process summary in
1451                 * a buffer, since we don't want to print anything out until
1452                 * after we release the process.
1453                 */
```



```
1455                /*
1456                 * The process is neither a system process nor defunct.
1457                 *
1458                 * Do printing and post-processing (like name lookups) after
1459                 * gathering the raw data from the process and releasing it.
1460                 * This way, we don't deadlock on (for example) name lookup
1461                 * if we grabbed the nscd and do 'pargs -x'.
1462                 *
1463                 * We always fetch the environment of the target, so that we
1464                 * can make an educated guess about its locale.
1465                 */
1466                get_env(&datap);
1467                if (aflag != 0)
1468                        get_args(&datap);
1469                if (xflag != 0)
1470                        get_auxv(&datap);

1472                /*
1473                 * If malloc() fails after this poiint, we return here to
1474                 * restore our locale and print a message.  If we don't
1475                 * reset this, we might erroneously try to Prelease a process
1476                 * twice.
1477                 */
1478                if ((r = setjmp(env)) != 0) {
1479                        (void) setlocale(LC_ALL, "");
1480                        (void) fprintf(stderr, "%s: out of memory: %s\n",
1481                            command, strerror(r));
1482                        return (1);
1483                }

1485                /*
1486                 * For the -l option, we need a proper name for this executable
1487                 * before we release it.
1488                 */
1489                if (lflag)
1490                        datap.pd_execname = Pexecname(Pr, execname,
1491                            sizeof (execname));

1493                Prelease(Pr, 0);

1495                /*
1496                 * Crawl through the environment to determine the locale of
1497                 * the target.
1498                 */
1499                lookup_locale(&datap);
1500                diflocale = 0;
1501                setup_conversions(&datap, &diflocale);

1503                if (lflag != 0) {
1504                        unprintable = 0;
1505                        convert_array(&datap, datap.pd_argv_strs,
1506                            datap.pd_argc, &unprintable);
1507                        if (diflocale)
1508                                (void) fprintf(stderr, "%s: Warning, target "
1509                                    "locale differs from current locale\n",
1510                                    command);
1511                        else if (unprintable)
1512                                (void) fprintf(stderr, "%s: Warning, command "
1513                                    "line contains unprintable characters\n",
1514                                    command);

1516                        retc += print_cmdline(&datap);
1517                } else {
1518                        psargs_conv = convert_str(&datap, psinfo.pr_psargs,
1519                            &unprintable);
```

```
1520                        info_sz = strlen(psargs_conv) + MAXPATHLEN + 32 + 1;
1521                        info = malloc(info_sz);
1522                        if (pstate == PS_DEAD) {
1523                                (void) snprintf(info, info_sz,
1524                                    "core '%s' of %d:\t%s\n",
1525                                    arg, (int)psinfo.pr_pid, psargs_conv);
1526                        } else {
1527                                (void) snprintf(info, info_sz, "%d:\t%s\n",
1528                                    (int)psinfo.pr_pid, psargs_conv);
1529                        }
1530                        (void) printf("%s", info);
1531                        free(info);
1532                        free(psargs_conv);

1534                        if (aflag != 0) {
1535                                convert_array(&datap, datap.pd_argv_strs,
1536                                    datap.pd_argc, &unprintable);
1537                                print_args(&datap);
1538                                if (eflag || xflag)
1539                                        (void) printf("\n");
1540                        }

1542                        if (eflag != 0) {
1543                                convert_array(&datap, datap.pd_envp_strs,
1544                                    datap.pd_envc, &unprintable);
1545                                print_env(&datap);
1546                                if (xflag)
1547                                        (void) printf("\n");
1548                        }

1550                        if (xflag != 0) {
1551                                convert_array(&datap, datap.pd_auxv_strs,
1552                                    datap.pd_auxc, &unprintable);
1553                                print_auxv(&datap);
1554                        }
1555                }

1557                cleanup_conversions(&datap);
1558                free_data(&datap);
1559        }

1561        return (retc != 0 ? 1 : 0);
1562 }
```

*unchanged_portion_omitted_*