```
**********************************************************
   38401 Tue Apr 26 13:38:45 2016
new/usr/src/cmd/ptools/pargs/pargs.c
6565 pargs crashes on growing env
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2007 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright (c) 2013, Joyent, Inc.  All rights reserved.
 */

/*
 * pargs examines and prints the arguments (argv), environment (environ),
 * and auxiliary vector of another process.
 *
 * This utility is made more complex because it must run in internationalized
 * environments.  The two key cases for pargs to manage are:
 *
 * 1. pargs and target run in the same locale: pargs must respect the
 * locale, but this case is straightforward.  Care is taken to correctly
 * use wide characters in order to print results properly.
 *
 * 2. pargs and target run in different locales: in this case, pargs examines
 * the string having assumed the victim's locale.  Unprintable (but valid)
 * characters are escaped.  Next, iconv(3c) is used to convert between the
 * target and pargs codeset.  Finally, a second pass to escape unprintable
 * (but valid) characters is made.
 *
 * In any case in which characters are encountered which are not valid in
 * their purported locale, the string "fails" and is treated as a traditional
 * 7-bit ASCII encoded string, and escaped accordingly.
 */

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <iconv.h>
#include <langinfo.h>
#include <unistd.h>
#include <ctype.h>
#include <fcntl.h>
#include <string.h>
#include <strings.h>
```



```
#include <limits.h>
#include <pwd.h>
#include <grp.h>
#include <errno.h>
#include <setjmp.h>
#include <sys/types.h>
#include <sys/auxv.h>
#include <sys/archsystm.h>
#include <sys/proc.h>
#include <sys/elf.h>
#include <libproc.h>
#include <wctype.h>
#include <widec.h>
#include <elfcap.h>

typedef struct pargs_data {
        struct ps_prochandle *pd_proc;  /* target proc handle */
        psinfo_t *pd_psinfo;            /* target psinfo */
        char *pd_locale;                /* target process locale */
        int pd_conv_flags;              /* flags governing string conversion */
        iconv_t pd_iconv;               /* iconv conversion descriptor */
        size_t pd_argc;
        uintptr_t *pd_argv;
        char **pd_argv_strs;
        size_t pd_envc;
        size_t pd_envc_curr;
#endif /* ! codereview */
        uintptr_t *pd_envp;
        char **pd_envp_strs;
        size_t pd_auxc;
        auxv_t *pd_auxv;
        char **pd_auxv_strs;
        char *pd_execname;
} pargs_data_t;

#define CONV_USE_ICONV          0x01
#define CONV_STRICT_ASCII       0x02

static char *command;
static int dmodel;

#define EXTRACT_BUFSZ 128               /* extract_string() initial size */
#define ENV_CHUNK 16                    /* #env ptrs to read at a time */

static jmp_buf env;                     /* malloc failure handling */

static void *
safe_zalloc(size_t size)
{
        void *p;

        /*
         * If the malloc fails we longjmp out to allow the code to Prelease()
         * a stopped victim if needed.
         */
        if ((p = malloc(size)) == NULL) {
                longjmp(env, errno);
        }

        bzero(p, size);
        return (p);
}

static char *
safe_strdup(const char *s1)
{
```

```
128         char    *s2;
129
130         s2 = safe_zalloc(strlen(s1) + 1);
131         (void) strcpy(s2, s1);
132         return (s2);
133 }
134
135 /*
136  * Given a wchar_t which might represent an 'escapable' sequence (see
137  * formats(5)), return the base ascii character needed to print that
138  * sequence.
139  *
140  * The comparisons performed may look suspect at first, but all are valid;
141  * the characters below all appear in the "Portable Character Set."  The
142  * Single Unix Spec says: "The wide-character value for each member of the
143  * Portable Character Set will equal its value when used as the lone
144  * character in an integer character constant."
145  */
146 static uchar_t
147 get_interp_char(wchar_t wc)
148 {
149         switch (wc) {
150         case L'\a':
151                 return ('a');
152         case L'\b':
153                 return ('b');
154         case L'\f':
155                 return ('f');
156         case L'\n':
157                 return ('n');
158         case L'\r':
159                 return ('r');
160         case L'\t':
161                 return ('t');
162         case L'\v':
163                 return ('v');
164         case L'\\':
165                 return ('\\');
166         }
167         return ('\0');
168 }
169
170 static char *
171 unctrl_str_strict_ascii(const char *src, int escape_slash, int *unprintable)
172 {
173         uchar_t *uc, *ucp, c, ic;
174         uc = ucp = safe_zalloc((strlen(src) * 4) + 1);
175         while ((c = *src++) != '\0') {
176                 /*
177                  * Call get_interp_char *first*, since \ will otherwise not
178                  * be escaped as \\.
179                  */
180                 if ((ic = get_interp_char((wchar_t)c)) != '\0') {
181                         if (escape_slash || ic != '\\')
182                                 *ucp++ = '\\';
183                         *ucp++ = ic;
184                 } else if (isascii(c) && isprint(c)) {
185                         *ucp++ = c;
186                 } else {
187                         *ucp++ = '\\';
188                         *ucp++ = ((c >> 6) & 7) + '0';
189                         *ucp++ = ((c >> 3) & 7) + '0';
190                         *ucp++ = (c & 7) + '0';
191                         *unprintable = 1;
192                 }
193         }
```



```
194         *ucp = '\0';
195         return ((char *)uc);
196 }
197
198 /*
199  * Convert control characters as described in format(5) to their readable
200  * representation; special care is taken to handle multibyte character sets.
201  *
202  * If escape_slash is true, escaping of '\' occurs.  The first time a string
203  * is unctrl'd, this should be '1'.  Subsequent iterations over the same
204  * string should set escape_slash to 0.  Otherwise you'll wind up with
205  * \ --> \\ --> \\\\.
206  */
207 static char *
208 unctrl_str(const char *src, int escape_slash, int *unprintable)
209 {
210         wchar_t wc;
211         wchar_t *wide_src, *wide_srcp;
212         wchar_t *wide_dest, *wide_destp;
213         char *uc;
214         size_t srcbufsz = strlen(src) + 1;
215         size_t destbufsz = srcbufsz * 4;
216         size_t srclen, destlen;
217
218         wide_srcp = wide_src = safe_zalloc(srcbufsz * sizeof (wchar_t));
219         wide_destp = wide_dest = safe_zalloc(destbufsz * sizeof (wchar_t));
220
221         if ((srclen = mbstowcs(wide_src, src, srcbufsz - 1)) == (size_t)-1) {
222                 /*
223                  * We can't trust the string, since in the locale in which
224                  * this call is operating, the string contains an invalid
225                  * multibyte sequence.  There isn't much to do here, so
226                  * convert the string byte by byte to wide characters, as
227                  * if it came from a C locale (char) string.  This isn't
228                  * perfect, but at least the characters will make it to
229                  * the screen.
230                  */
231                 free(wide_src);
232                 free(wide_dest);
233                 return (unctrl_str_strict_ascii(src, escape_slash,
234                     unprintable));
235         }
236         if (srclen == (srcbufsz - 1)) {
237                 wide_src[srclen] = L'\0';
238         }
239
240         while ((wc = *wide_srcp++) != L'\0') {
241                 char cvt_buf[MB_LEN_MAX];
242                 int len, i;
243                 char c = get_interp_char(wc);
244
245                 if ((c != '\0') && (escape_slash || c != '\\')) {
246                         /*
247                          * Print "interpreted version" (\n, \a, etc).
248                          */
249                         *wide_destp++ = L'\\';
250                         *wide_destp++ = (wchar_t)c;
251                         continue;
252                 }
253
254                 if (iswprint(wc)) {
255                         *wide_destp++ = wc;
256                         continue;
257                 }
258
259                 /*
```

```
260                  * Convert the wide char back into (potentially several)
261                  * multibyte characters, then escape out each of those bytes.
262                  */
263                 bzero(cvt_buf, sizeof (cvt_buf));
264                 if ((len = wctomb(cvt_buf, wc)) == -1) {
265                         /*
266                          * This is a totally invalid wide char; discard it.
267                          */
268                         continue;
269                 }
270                 for (i = 0; i < len; i++) {
271                         uchar_t c = cvt_buf[i];
272                         *wide_destp++ = L'\\';
273                         *wide_destp++ = (wchar_t)('0' + ((c >> 6) & 7));
274                         *wide_destp++ = (wchar_t)('0' + ((c >> 3) & 7));
275                         *wide_destp++ = (wchar_t)('0' + (c & 7));
276                         *unprintable = 1;
277                 }
278         }
279
280         *wide_destp = '\0';
281         destlen = (wide_destp - wide_dest) * MB_CUR_MAX + 1;
282         uc = safe_zalloc(destlen);
283         if (wcstombs(uc, wide_dest, destlen) == (size_t)-1) {
284                 /* If we've gotten this far, wcstombs shouldn't fail... */
285                 (void) fprintf(stderr, "%s: wcstombs failed unexpectedly: %s\n",
286                     command, strerror(errno));
287                 exit(1);
288         } else {
289                 char *tmp;
290                 /*
291                  * Try to save memory; don't waste 3 * strlen in the
292                  * common case.
293                  */
294                 tmp = safe_strdup(uc);
295                 free(uc);
296                 uc = tmp;
297         }
298         free(wide_dest);
299         free(wide_src);
300         return (uc);
301 }
302
303 /*
304  * These functions determine which characters are safe to be left unquoted.
305  * Rather than starting with every printable character and subtracting out the
306  * shell metacharacters, we take the more conservative approach of starting with
307  * a set of safe characters and adding those few common punctuation characters
308  * which are known to be safe.  The rules are:
309  *
310  *      If this is a printable character (graph), and not punctuation, it is
311  *      safe to leave unquoted.
312  *
313  *      If it's one of known hard-coded safe characters, it's also safe to leave
314  *      unquoted.
315  *
316  *      Otherwise, the entire argument must be quoted.
317  *
318  * This will cause some strings to be unecessarily quoted, but it is safer than
319  * having a character unintentionally interpreted by the shell.
320  */
321 static int
322 issafe_ascii(char c)
323 {
324         return (isalnum(c) || strchr("_.-/@:,", c) != NULL);
325 }
```



```
327 static int
328 issafe(wchar_t wc)
329 {
330         return ((iswgraph(wc) && !iswpunct(wc)) ||
331             wschr(L"_.-/@:,", wc) != NULL);
332 }
333
334 /*ARGSUSED*/
335 static char *
336 quote_string_ascii(pargs_data_t *datap, char *src)
337 {
338         char *dst;
339         int quote_count = 0;
340         int need_quote = 0;
341         char *srcp, *dstp;
342         size_t dstlen;
343
344         for (srcp = src; *srcp != '\0'; srcp++) {
345                 if (!issafe_ascii(*srcp)) {
346                         need_quote = 1;
347                         if (*srcp == '\'')
348                                 quote_count++;
349                 }
350         }
351
352         if (!need_quote)
353                 return (src);
354
355         /*
356          * The only character we care about here is a single quote.  All the
357          * other unprintable characters (and backslashes) will have been dealt
358          * with by unctrl_str().  We make the following subtitution when we
359          * encounter a single quote:
360          *
361          *      ' = '"'"'
362          *
363          * In addition, we put single quotes around the entire argument.  For
364          * example:
365          *
366          *      foo'bar = 'foo'"'"'bar'
367          */
368         dstlen = strlen(src) + 3 + 4 * quote_count;
369         dst = safe_zalloc(dstlen);
370
371         dstp = dst;
372         *dstp++ = '\'';
373         for (srcp = src; *srcp != '\0'; srcp++, dstp++) {
374                 *dstp = *srcp;
375
376                 if (*srcp == '\'') {
377                         dstp[1] = '"';
378                         dstp[2] = '\'';
379                         dstp[3] = '"';
380                         dstp[4] = '\'';
381                         dstp += 4;
382                 }
383         }
384         *dstp++ = '\'';
385         *dstp = '\0';
386
387         free(src);
388
389         return (dst);
390 }
```

```
392 static char *
393 quote_string(pargs_data_t *datap, char *src)
394 {
395         wchar_t *wide_src, *wide_srcp;
396         wchar_t *wide_dest, *wide_destp;
397         char *uc;
398         size_t srcbufsz = strlen(src) + 1;
399         size_t srclen;
400         size_t destbufsz;
401         size_t destlen;
402         int quote_count = 0;
403         int need_quote = 0;

405         if (datap->pd_conv_flags & CONV_STRICT_ASCII)
406                 return (quote_string_ascii(datap, src));

408         wide_srcp = wide_src = safe_zalloc(srcbufsz * sizeof (wchar_t));

410         if ((srclen = mbstowcs(wide_src, src, srcbufsz - 1)) == (size_t)-1) {
411                 free(wide_src);
412                 return (quote_string_ascii(datap, src));
413         }

415         if (srclen == srcbufsz - 1)
416                 wide_src[srclen] = L'\0';

418         for (wide_srcp = wide_src; *wide_srcp != '\0'; wide_srcp++) {
419                 if (!issafe(*wide_srcp)) {
420                         need_quote = 1;
421                         if (*wide_srcp == L'\'')
422                                 quote_count++;
423                 }
424         }

426         if (!need_quote) {
427                 free(wide_src);
428                 return (src);
429         }

431         /*
432          * See comment for quote_string_ascii(), above.
433          */
434         destbufsz = srcbufsz + 3 + 4 * quote_count;
435         wide_destp = wide_dest = safe_zalloc(destbufsz * sizeof (wchar_t));

437         *wide_destp++ = L'\'';
438         for (wide_srcp = wide_src; *wide_srcp != L'\0';
439             wide_srcp++, wide_destp++) {
440                 *wide_destp = *wide_srcp;

442                 if (*wide_srcp == L'\'') {
443                         wide_destp[1] = L'"';
444                         wide_destp[2] = L'\'';
445                         wide_destp[3] = L'"';
446                         wide_destp[4] = L'\'';
447                         wide_destp += 4;
448                 }
449         }
450         *wide_destp++ = L'\'';
451         *wide_destp = L'\0';

453         destlen = destbufsz * MB_CUR_MAX + 1;
454         uc = safe_zalloc(destlen);
455         if (wcstombs(uc, wide_dest, destlen) == (size_t)-1) {
456                 /* If we've gotten this far, wcstombs shouldn't fail... */
457                 (void) fprintf(stderr, "%s: wcstombs failed unexpectedly: %s\n",
```



```
458                     command, strerror(errno));
459                 exit(1);
460         }

462         free(wide_dest);
463         free(wide_src);

465         return (uc);
466 }


469 /*
470  * Determine the locale of the target process by traversing its environment,
471  * making only one pass for efficiency's sake; stash the result in
472  * datap->pd_locale.
473  *
474  * It's possible that the process has called setlocale() to change its
475  * locale to something different, but we mostly care about making a good
476  * guess as to the locale at exec(2) time.
477  */
478 static void
479 lookup_locale(pargs_data_t *datap)
480 {
481         int i, j, composite = 0;
482         size_t  len = 0;
483         char    *pd_locale;
484         char    *lc_all = NULL, *lang = NULL;
485         char    *lcs[] = { NULL, NULL, NULL, NULL, NULL, NULL };
486         static const char *cat_names[] = {
487                 "LC_CTYPE=",    "LC_NUMERIC=",  "LC_TIME=",
488                 "LC_COLLATE=",  "LC_MONETARY=", "LC_MESSAGES="
489         };

491         for (i = 0; i < datap->pd_envc; i++) {
492                 char *s = datap->pd_envp_strs[i];

494                 if (s == NULL)
495                         continue;

497                 if (strncmp("LC_ALL=", s, strlen("LC_ALL=")) == 0) {
498                         /*
499                          * Minor optimization-- if we find LC_ALL we're done.
500                          */
501                         lc_all = s + strlen("LC_ALL=");
502                         break;
503                 }
504                 for (j = 0; j <= _LastCategory; j++) {
505                         if (strncmp(cat_names[j], s,
506                             strlen(cat_names[j])) == 0) {
507                                 lcs[j] = s + strlen(cat_names[j]);
508                         }
509                 }
510                 if (strncmp("LANG=", s, strlen("LANG=")) == 0) {
511                         lang = s + strlen("LANG=");
512                 }
513         }

515         if (lc_all && (*lc_all == '\0'))
516                 lc_all = NULL;
517         if (lang && (*lang == '\0'))
518                 lang = NULL;

520         for (i = 0; i <= _LastCategory; i++) {
521                 if (lc_all != NULL) {
522                         lcs[i] = lc_all;
523                 } else if (lcs[i] != NULL) {
```

```
524                         lcs[i] = lcs[i];
525                 } else if (lang != NULL) {
526                         lcs[i] = lang;
527                 } else {
528                         lcs[i] = "C";
529                 }
530                 if ((i > 0) && (lcs[i] != lcs[i-1]))
531                         composite++;

533                 len += 1 + strlen(lcs[i]);      /* 1 extra byte for '/' */
534         }

536         if (composite == 0) {
537                 /* simple locale */
538                 pd_locale = safe_strdup(lcs[0]);
539         } else {
540                 /* composite locale */
541                 pd_locale = safe_zalloc(len + 1);
542                 (void) snprintf(pd_locale, len + 1, "/%s/%s/%s/%s/%s/%s",
543                     lcs[0], lcs[1], lcs[2], lcs[3], lcs[4], lcs[5]);
544         }
545         datap->pd_locale = pd_locale;
546 }

548 /*
549  * Pull a string from the victim, regardless of size; this routine allocates
550  * memory for the string which must be freed by the caller.
551  */
552 static char *
553 extract_string(pargs_data_t *datap, uintptr_t addr)
554 {
555         int size = EXTRACT_BUFSZ;
556         char *result;

558         result = safe_zalloc(size);

560         for (;;) {
561                 if (Pread_string(datap->pd_proc, result, size, addr) < 0) {
562                         free(result);
563                         return (NULL);
564                 } else if (strlen(result) == (size - 1)) {
565                         free(result);
566                         size *= 2;
567                         result = safe_zalloc(size);
568                 } else {
569                         break;
570                 }
571         }
572         return (result);
573 }

575 /*
576  * Utility function to read an array of pointers from the victim, adjusting
577  * for victim data model; returns the number of bytes successfully read.
578  */
579 static ssize_t
580 read_ptr_array(pargs_data_t *datap, uintptr_t offset, uintptr_t *buf,
581     size_t nelems)
582 {
583         ssize_t res;

585         if (dmodel == PR_MODEL_NATIVE) {
586                 res = Pread(datap->pd_proc, buf, nelems * sizeof (uintptr_t),
587                     offset);
588         } else {
589                 int i;
```



```
590                 uint32_t *arr32 = safe_zalloc(nelems * sizeof (uint32_t));

592                 res = Pread(datap->pd_proc, arr32, nelems * sizeof (uint32_t),
593                     offset);
594                 if (res > 0) {
595                         for (i = 0; i < nelems; i++)
596                                 buf[i] = arr32[i];
597                 }
598                 free(arr32);
599         }
600         return (res);
601 }

603 /*
604  * Extract the argv array from the victim; store the pointer values in
605  * datap->pd_argv and the extracted strings in datap->pd_argv_strs.
606  */
607 static void
608 get_args(pargs_data_t *datap)
609 {
610         size_t argc = datap->pd_psinfo->pr_argc;
611         uintptr_t argvoff = datap->pd_psinfo->pr_argv;
612         int i;

614         datap->pd_argc = argc;
615         datap->pd_argv = safe_zalloc(argc * sizeof (uintptr_t));

617         if (read_ptr_array(datap, argvoff, datap->pd_argv, argc) <= 0) {
618                 free(datap->pd_argv);
619                 datap->pd_argv = NULL;
620                 return;
621         }

623         datap->pd_argv_strs = safe_zalloc(argc * sizeof (char *));
624         for (i = 0; i < argc; i++) {
625                 if (datap->pd_argv[i] == 0)
626                         continue;
627                 datap->pd_argv_strs[i] = extract_string(datap,
628                     datap->pd_argv[i]);
629         }
630 }

632 /*ARGSUSED*/
633 static int
634 build_env(void *data, struct ps_prochandle *pr, uintptr_t addr, const char *str)
635 {
636         pargs_data_t *datap = data;

638         if (datap->pd_envp != NULL) {
639                 /* env has more items than last time, skip the newer ones */
640                 if (datap->pd_envc > datap->pd_envc_curr)
641                         return (0);

643 #endif /* ! codereview */
644                 datap->pd_envp[datap->pd_envc] = addr;
645                 if (str == NULL)
646                         datap->pd_envp_strs[datap->pd_envc] = NULL;
647                 else
648                         datap->pd_envp_strs[datap->pd_envc] = strdup(str);
649         }

651         datap->pd_envc++;

653         return (0);
654 }
```

```
656 static void
657 get_env(pargs_data_t *datap)
658 {
659         struct ps_prochandle *pr = datap->pd_proc;

661         datap->pd_envc = 0;
662         (void) Penv_iter(pr, build_env, datap);
663         datap->pd_envc_curr = datap->pd_envc;
664 #endif /* ! codereview */

666         datap->pd_envp = safe_zalloc(sizeof (uintptr_t) * datap->pd_envc);
667         datap->pd_envp_strs = safe_zalloc(sizeof (char *) * datap->pd_envc);

669         datap->pd_envc = 0;
670         (void) Penv_iter(pr, build_env, datap);
671 }

673 /*
674  * The following at_* routines are used to decode data from the aux vector.
675  */

677 /*ARGSUSED*/
678 static void
679 at_null(long val, char *instr, size_t n, char *str)
680 {
681         str[0] = '\0';
682 }

684 /*ARGSUSED*/
685 static void
686 at_str(long val, char *instr, size_t n, char *str)
687 {
688         str[0] = '\0';
689         if (instr != NULL) {
690                 (void) strlcpy(str, instr, n);
691         }
692 }

694 /*
695  * Note: Don't forget to add a corresponding case to isainfo(1).
696  */

698 #define FMT_AV(s, n, hwcap, mask, name)                         \
699         if ((hwcap) & (mask))                                   \
700                 (void) snprintf(s, n, "%s" name " | ", s)

702 /*ARGSUSED*/
703 static void
704 at_hwcap(long val, char *instr, size_t n, char *str)
705 {
706 #if defined(__sparc) || defined(__sparcv9)
707         (void) elfcap_hw1_to_str(ELFCAP_STYLE_UC, val, str, n,
708             ELFCAP_FMT_PIPSPACE, EM_SPARC);

710 #elif defined(__i386) || defined(__amd64)
711         (void) elfcap_hw1_to_str(ELFCAP_STYLE_UC, val, str, n,
712             ELFCAP_FMT_PIPSPACE, EM_386);
713 #else
714 #error  "port me"
715 #endif
716 }

718 /*ARGSUSED*/
719 static void
720 at_hwcap2(long val, char *instr, size_t n, char *str)
721 {
```



```
722 #if defined(__sparc) || defined(__sparcv9)
723         (void) elfcap_hw2_to_str(ELFCAP_STYLE_UC, val, str, n,
724             ELFCAP_FMT_PIPSPACE, EM_SPARC);

726 #elif defined(__i386) || defined(__amd64)
727         (void) elfcap_hw2_to_str(ELFCAP_STYLE_UC, val, str, n,
728             ELFCAP_FMT_PIPSPACE, EM_386);
729 #else
730 #error  "port me"
731 #endif
732 }


735 /*ARGSUSED*/
736 static void
737 at_uid(long val, char *instr, size_t n, char *str)
738 {
739         struct passwd *pw = getpwuid((uid_t)val);

741         if ((pw == NULL) || (pw->pw_name == NULL))
742                 str[0] = '\0';
743         else
744                 (void) snprintf(str, n, "%lu(%s)", val, pw->pw_name);
745 }


748 /*ARGSUSED*/
749 static void
750 at_gid(long val, char *instr, size_t n, char *str)
751 {
752         struct group *gr = getgrgid((gid_t)val);

754         if ((gr == NULL) || (gr->gr_name == NULL))
755                 str[0] = '\0';
756         else
757                 (void) snprintf(str, n, "%lu(%s)", val, gr->gr_name);
758 }

760 static struct auxfl {
761         int af_flag;
762         const char *af_name;
763 } auxfl[] = {
764         { AF_SUN_SETUGID,       "setugid" },
765 };

767 /*ARGSUSED*/
768 static void
769 at_flags(long val, char *instr, size_t n, char *str)
770 {
771         int i;

773         *str = '\0';

775         for (i = 0; i < sizeof (auxfl)/sizeof (struct auxfl); i++) {
776                 if ((val & auxfl[i].af_flag) != 0) {
777                         if (*str != '\0')
778                                 (void) strlcat(str, ",", n);
779                         (void) strlcat(str, auxfl[i].af_name, n);
780                 }
781         }
782 }

784 #define MAX_AT_NAME_LEN 15

786 struct aux_id {
787         int aux_type;
```

```
788         const char *aux_name;
789         void (*aux_decode)(long, char *, size_t, char *);
790 };

792 static struct aux_id aux_arr[] = {
793         { AT_NULL,              "AT_NULL",              at_null },
794         { AT_IGNORE,            "AT_IGNORE",            at_null },
795         { AT_EXECFD,            "AT_EXECFD",            at_null },
796         { AT_PHDR,              "AT_PHDR",              at_null },
797         { AT_PHENT,             "AT_PHENT",             at_null },
798         { AT_PHNUM,             "AT_PHNUM",             at_null },
799         { AT_PAGESZ,            "AT_PAGESZ",            at_null },
800         { AT_BASE,              "AT_BASE",              at_null },
801         { AT_FLAGS,             "AT_FLAGS",             at_null },
802         { AT_ENTRY,             "AT_ENTRY",             at_null },
803         { AT_SUN_UID,           "AT_SUN_UID",           at_uid  },
804         { AT_SUN_RUID,          "AT_SUN_RUID",          at_uid  },
805         { AT_SUN_GID,           "AT_SUN_GID",           at_gid  },
806         { AT_SUN_RGID,          "AT_SUN_RGID",          at_gid  },
807         { AT_SUN_LDELF,         "AT_SUN_LDELF",         at_null },
808         { AT_SUN_LDSHDR,        "AT_SUN_LDSHDR",        at_null },
809         { AT_SUN_LDNAME,        "AT_SUN_LDNAME",        at_null },
810         { AT_SUN_LPAGESZ,       "AT_SUN_LPAGESZ",       at_null },
811         { AT_SUN_PLATFORM,      "AT_SUN_PLATFORM",      at_str  },
812         { AT_SUN_EXECNAME,      "AT_SUN_EXECNAME",      at_str  },
813         { AT_SUN_HWCAP,         "AT_SUN_HWCAP",         at_hwcap },
814         { AT_SUN_HWCAP2,        "AT_SUN_HWCAP2",        at_hwcap2 },
815         { AT_SUN_IFLUSH,        "AT_SUN_IFLUSH",        at_null },
816         { AT_SUN_CPU,           "AT_SUN_CPU",           at_null },
817         { AT_SUN_MMU,           "AT_SUN_MMU",           at_null },
818         { AT_SUN_LDDATA,        "AT_SUN_LDDATA",        at_null },
819         { AT_SUN_AUXFLAGS,      "AT_SUN_AUXFLAGS",      at_flags },
820         { AT_SUN_EMULATOR,      "AT_SUN_EMULATOR",      at_str  },
821         { AT_SUN_BRANDNAME,     "AT_SUN_BRANDNAME",     at_str  },
822         { AT_SUN_BRAND_AUX1,    "AT_SUN_BRAND_AUX1",    at_null },
823         { AT_SUN_BRAND_AUX2,    "AT_SUN_BRAND_AUX2",    at_null },
824         { AT_SUN_BRAND_AUX3,    "AT_SUN_BRAND_AUX3",    at_null }
825 };

827 #define N_AT_ENTS (sizeof (aux_arr) / sizeof (struct aux_id))

829 /*
830  * Return the aux_id entry for the given aux type; returns NULL if not found.
831  */
832 static struct aux_id *
833 aux_find(int type)
834 {
835         int i;

837         for (i = 0; i < N_AT_ENTS; i++) {
838                 if (type == aux_arr[i].aux_type)
839                         return (&aux_arr[i]);
840         }

842         return (NULL);
843 }

845 static void
846 get_auxv(pargs_data_t *datap)
847 {
848         int i;
849         const auxv_t *auxvp;

851         /*
852          * Fetch the aux vector from the target process.
853          */
```



```
854         if (ps_pauxv(datap->pd_proc, &auxvp) != PS_OK)
855                 return;

857         for (i = 0; auxvp[i].a_type != AT_NULL; i++)
858                 continue;

860         datap->pd_auxc = i;
861         datap->pd_auxv = safe_zalloc(i * sizeof (auxv_t));
862         bcopy(auxvp, datap->pd_auxv, i * sizeof (auxv_t));

864         datap->pd_auxv_strs = safe_zalloc(datap->pd_auxc * sizeof (char *));
865         for (i = 0; i < datap->pd_auxc; i++) {
866                 struct aux_id *aux = aux_find(datap->pd_auxv[i].a_type);

868                 /*
869                  * Grab strings for those entries which have a string-decoder.
870                  */
871                 if ((aux != NULL) && (aux->aux_decode == at_str)) {
872                         datap->pd_auxv_strs[i] =
873                             extract_string(datap, datap->pd_auxv[i].a_un.a_val);
874                 }
875         }
876 }

878 /*
879  * Prepare to convert characters in the victim's character set into user's
880  * character set.
881  */
882 static void
883 setup_conversions(pargs_data_t *datap, int *diflocale)
884 {
885         char *mylocale = NULL, *mycharset = NULL;
886         char *targetlocale = NULL, *targetcharset = NULL;

888         mycharset = safe_strdup(nl_langinfo(CODESET));

890         mylocale = setlocale(LC_CTYPE, NULL);
891         if ((mylocale == NULL) || (strcmp(mylocale, "") == 0))
892                 mylocale = "C";
893         mylocale = safe_strdup(mylocale);

895         if (datap->pd_conv_flags & CONV_STRICT_ASCII)
896                 goto done;

898         /*
899          * If the target's locale is "C" or "POSIX", go fast.
900          */
901         if ((strcmp(datap->pd_locale, "C") == 0) ||
902             (strcmp(datap->pd_locale, "POSIX") == 0)) {
903                 datap->pd_conv_flags |= CONV_STRICT_ASCII;
904                 goto done;
905         }

907         /*
908          * Switch to the victim's locale, and discover its character set.
909          */
910         if (setlocale(LC_ALL, datap->pd_locale) == NULL) {
911                 (void) fprintf(stderr,
912                     "%s: Couldn't determine locale of target process.\n",
913                     command);
914                 (void) fprintf(stderr,
915                     "%s: Some strings may not be displayed properly.\n",
916                     command);
917                 goto done;
918         }
```

```
920         /*
921          * Get LC_CTYPE part of target's locale, and its codeset.
922          */
923         targetlocale = safe_strdup(setlocale(LC_CTYPE, NULL));
924         targetcharset = safe_strdup(nl_langinfo(CODESET));

926         /*
927          * Now go fully back to the pargs user's locale.
928          */
929         (void) setlocale(LC_ALL, "");

931         /*
932          * It's safe to bail here if the lc_ctype of the locales are the
933          * same-- we know that their encodings and characters sets are the same.
934          */
935         if (strcmp(targetlocale, mylocale) == 0)
936                 goto done;

938         *diflocale = 1;

940         /*
941          * If the codeset of the victim matches our codeset then iconv need
942          * not be involved.
943          */
944         if (strcmp(mycharset, targetcharset) == 0)
945                 goto done;

947         if ((datap->pd_iconv = iconv_open(mycharset, targetcharset))
948             == (iconv_t)-1) {
949                 /*
950                  * EINVAL indicates there was no conversion available
951                  * from victim charset to mycharset
952                  */
953                 if (errno != EINVAL) {
954                         (void) fprintf(stderr,
955                             "%s: failed to initialize iconv: %s\n",
956                             command, strerror(errno));
957                         exit(1);
958                 }
959                 datap->pd_conv_flags |= CONV_STRICT_ASCII;
960         } else {
961                 datap->pd_conv_flags |= CONV_USE_ICONV;
962         }
963 done:
964         free(mycharset);
965         free(mylocale);
966         free(targetcharset);
967         free(targetlocale);
968 }

970 static void
971 cleanup_conversions(pargs_data_t *datap)
972 {
973         if (datap->pd_conv_flags & CONV_USE_ICONV) {
974                 (void) iconv_close(datap->pd_iconv);
975         }
976 }

978 static char *
979 convert_run_iconv(pargs_data_t *datap, const char *str)
980 {
981         size_t inleft, outleft, bufsz = 64;
982         char *outstr, *outstrptr;
983         const char *instrptr;

985         for (;;) {
```



```
986                 outstrptr = outstr = safe_zalloc(bufsz + 1);
987                 outleft = bufsz;

989                 /*
990                  * Generate the "initial shift state" sequence, placing that
991                  * at the head of the string.
992                  */
993                 inleft = 0;
994                 (void) iconv(datap->pd_iconv, NULL, &inleft,
995                     &outstrptr, &outleft);

997                 inleft = strlen(str);
998                 instrptr = str;
999                 if (iconv(datap->pd_iconv, &instrptr, &inleft, &outstrptr,
1000                    &outleft) != (size_t)-1) {
1001                        /*
1002                         * Outstr must be null terminated upon exit from
1003                         * iconv().
1004                         */
1005                        *(outstr + (bufsz - outleft)) = '\0';
1006                        break;
1007                } else if (errno == E2BIG) {
1008                        bufsz *= 2;
1009                        free(outstr);
1010                } else if ((errno == EILSEQ) || (errno == EINVAL)) {
1011                        free(outstr);
1012                        return (NULL);
1013                } else {
1014                        /*
1015                         * iconv() could in theory return EBADF, but that
1016                         * shouldn't happen.
1017                         */
1018                        (void) fprintf(stderr,
1019                            "%s: iconv(3C) failed unexpectedly: %s\n",
1020                            command, strerror(errno));

1022                        exit(1);
1023                }
1024        }
1025        return (outstr);
1026 }

1028 /*
1029  * Returns a freshly allocated string converted to the local character set,
1030  * removed of unprintable characters.
1031  */
1032 static char *
1033 convert_str(pargs_data_t *datap, const char *str, int *unprintable)
1034 {
1035         char *retstr, *tmp;

1037         if (datap->pd_conv_flags & CONV_STRICT_ASCII) {
1038                 retstr = unctrl_str_strict_ascii(str, 1, unprintable);
1039                 return (retstr);
1040         }

1042         if ((datap->pd_conv_flags & CONV_USE_ICONV) == 0) {
1043                 /*
1044                  * If we aren't using iconv(), convert control chars in
1045                  * the string in pargs' locale, since that is the display
1046                  * locale.
1047                  */
1048                 retstr = unctrl_str(str, 1, unprintable);
1049                 return (retstr);
1050         }
```

```
1052         /*
1053          * The logic here is a bit (ahem) tricky.  Start by converting
1054          * unprintable characters *in the target's locale*.  This should
1055          * eliminate a variety of unprintable or illegal characters-- in
1056          * short, it should leave us with something which iconv() won't
1057          * have trouble with.
1058          *
1059          * After allowing iconv to convert characters as needed, run unctrl
1060          * again in pargs' locale-- This time to make sure that any
1061          * characters which aren't printable according to the *current*
1062          * locale (independent of the current codeset) get taken care of.
1063          * Without this second stage, we might (for example) fail to
1064          * properly handle characters converted into the 646 character set
1065          * (which are 8-bits wide), but which must be displayed in the C
1066          * locale (which uses 646, but whose printable characters are a
1067          * subset of the 7-bit characters).
1068          *
1069          * Note that assuming the victim's locale using LC_ALL will be
1070          * problematic when pargs' messages are internationalized in the
1071          * future (and it calls textdomain(3C)).  In this case, any
1072          * error message fprintf'd in unctrl_str() will be in the wrong
1073          * LC_MESSAGES class.  We'll cross that bridge when we come to it.
1074          */
1075         (void) setlocale(LC_ALL, datap->pd_locale);
1076         retstr = unctrl_str(str, 1, unprintable);
1077         (void) setlocale(LC_ALL, "");

1079         tmp = retstr;
1080         if ((retstr = convert_run_iconv(datap, retstr)) == NULL) {
1081                 /*
1082                  * In this (rare but real) case, the iconv() failed even
1083                  * though we unctrl'd the string.  Treat the original string
1084                  * (str) as a C locale string and strip it that way.
1085                  */
1086                 free(tmp);
1087                 return (unctrl_str_strict_ascii(str, 0, unprintable));
1088         }

1090         free(tmp);
1091         tmp = retstr;
1092         /*
1093          * Run unctrl_str, but make sure not to escape \ characters, which
1094          * may have resulted from the first round of unctrl.
1095          */
1096         retstr = unctrl_str(retstr, 0, unprintable);
1097         free(tmp);
1098         return (retstr);
1099 }


1102 static void
1103 convert_array(pargs_data_t *datap, char **arr, size_t count, int *unprintable)
1104 {
1105         int i;
1106         char *tmp;

1108         if (arr == NULL)
1109                 return;

1111         for (i = 0; i < count; i++) {
1112                 if ((tmp = arr[i]) == NULL)
1113                         continue;
1114                 arr[i] = convert_str(datap, arr[i], unprintable);
1115                 free(tmp);
1116         }
1117 }
```



```
1119 /*
1120  * Free data allocated during the gathering phase.
1121  */
1122 static void
1123 free_data(pargs_data_t *datap)
1124 {
1125         int i;

1127         if (datap->pd_argv) {
1128                 for (i = 0; i < datap->pd_argc; i++) {
1129                         if (datap->pd_argv_strs[i] != NULL)
1130                                 free(datap->pd_argv_strs[i]);
1131                 }
1132                 free(datap->pd_argv);
1133                 free(datap->pd_argv_strs);
1134         }

1136         if (datap->pd_envp) {
1137                 for (i = 0; i < datap->pd_envc; i++) {
1138                         if (datap->pd_envp_strs[i] != NULL)
1139                                 free(datap->pd_envp_strs[i]);
1140                 }
1141                 free(datap->pd_envp);
1142                 free(datap->pd_envp_strs);
1143         }

1145         if (datap->pd_auxv) {
1146                 for (i = 0; i < datap->pd_auxc; i++) {
1147                         if (datap->pd_auxv_strs[i] != NULL)
1148                                 free(datap->pd_auxv_strs[i]);
1149                 }
1150                 free(datap->pd_auxv);
1151                 free(datap->pd_auxv_strs);
1152         }
1153 }

1155 static void
1156 print_args(pargs_data_t *datap)
1157 {
1158         int i;

1160         if (datap->pd_argv == NULL) {
1161                 (void) fprintf(stderr, "%s: failed to read argv[]\n", command);
1162                 return;
1163         }

1165         for (i = 0; i < datap->pd_argc; i++) {
1166                 (void) printf("argv[%d]: ", i);
1167                 if (datap->pd_argv[i] == NULL) {
1168                         (void) printf("<NULL>\n");
1169                 } else if (datap->pd_argv_strs[i] == NULL) {
1170                         (void) printf("<0x%0*lx>\n",
1171                             (dmodel == PR_MODEL_LP64)? 16 : 8,
1172                             (long)datap->pd_argv[i]);
1173                 } else {
1174                         (void) printf("%s\n", datap->pd_argv_strs[i]);
1175                 }
1176         }
1177 }

1179 static void
1180 print_env(pargs_data_t *datap)
1181 {
1182         int i;
```

```
1184         if (datap->pd_envp == NULL) {
1185                 (void) fprintf(stderr, "%s: failed to read envp[]\n", command);
1186                 return;
1187         }

1189         for (i = 0; i < datap->pd_envc; i++) {
1190                 (void) printf("envp[%d]: ", i);
1191                 if (datap->pd_envp[i] == 0) {
1192                         break;
1193                 } else if (datap->pd_envp_strs[i] == NULL) {
1194                         (void) printf("<0x%0*lx>\n",
1195                             (dmodel == PR_MODEL_LP64)? 16 : 8,
1196                             (long)datap->pd_envp[i]);
1197                 } else {
1198                         (void) printf("%s\n", datap->pd_envp_strs[i]);
1199                 }
1200         }
1201 }

1203 static int
1204 print_cmdline(pargs_data_t *datap)
1205 {
1206         int i;

1208         /*
1209          * Go through and check to see if we have valid data.  If not, print
1210          * an error message and bail.
1211          */
1212         for (i = 0; i < datap->pd_argc; i++) {
1213                 if (datap->pd_argv == NULL || datap->pd_argv[i] == NULL ||
1214                     datap->pd_argv_strs[i] == NULL) {
1215                         (void) fprintf(stderr, "%s: target has corrupted "
1216                             "argument list\n", command);
1217                         return (1);
1218                 }

1220                 datap->pd_argv_strs[i] =
1221                     quote_string(datap, datap->pd_argv_strs[i]);
1222         }

1224         if (datap->pd_execname == NULL) {
1225                 (void) fprintf(stderr, "%s: cannot determine name of "
1226                     "executable\n", command);
1227                 return (1);
1228         }

1230         (void) printf("%s ", datap->pd_execname);

1232         for (i = 1; i < datap->pd_argc; i++)
1233                 (void) printf("%s ", datap->pd_argv_strs[i]);

1235         (void) printf("\n");

1237         return (0);
1238 }

1240 static void
1241 print_auxv(pargs_data_t *datap)
1242 {
1243         int i;
1244         const auxv_t *pa;

1246         /*
1247          * Print the names and values of all the aux vector entries.
1248          */
1249         for (i = 0; i < datap->pd_auxc; i++) {
```



```
1250                 char type[32];
1251                 char decode[PATH_MAX];
1252                 struct aux_id *aux;
1253                 long v;
1254                 pa = &datap->pd_auxv[i];

1256                 aux = aux_find(pa->a_type);
1257                 v = (long)pa->a_un.a_val;

1259                 if (aux != NULL) {
1260                         /*
1261                          * Fetch aux vector type string and decoded
1262                          * representation of the value.
1263                          */
1264                         (void) strlcpy(type, aux->aux_name, sizeof (type));
1265                         aux->aux_decode(v, datap->pd_auxv_strs[i],
1266                             sizeof (decode), decode);
1267                 } else {
1268                         (void) snprintf(type, sizeof (type), "%d", pa->a_type);
1269                         decode[0] = '\0';
1270                 }

1272                 (void) printf("%-*s 0x%0*lx %s\n", MAX_AT_NAME_LEN, type,
1273                     (dmodel == PR_MODEL_LP64)? 16 : 8, v, decode);
1274         }
1275 }

1277 int
1278 main(int argc, char *argv[])
1279 {
1280         int aflag = 0, cflag = 0, eflag = 0, xflag = 0, lflag = 0;
1281         int errflg = 0, retc = 0;
1282         int opt;
1283         int error = 1;
1284         core_content_t content = 0;

1286         (void) setlocale(LC_ALL, "");

1288         if ((command = strrchr(argv[0], '/')) != NULL)
1289                 command++;
1290         else
1291                 command = argv[0];

1293         while ((opt = getopt(argc, argv, "acelxF")) != EOF) {
1294                 switch (opt) {
1295                 case 'a':               /* show process arguments */
1296                         content |= CC_CONTENT_STACK;
1297                         aflag++;
1298                         break;
1299                 case 'c':               /* force 7-bit ascii */
1300                         cflag++;
1301                         break;
1302                 case 'e':               /* show environment variables */
1303                         content |= CC_CONTENT_STACK;
1304                         eflag++;
1305                         break;
1306                 case 'l':
1307                         lflag++;
1308                         aflag++;        /* -l implies -a */
1309                         break;
1310                 case 'x':               /* show aux vector entries */
1311                         xflag++;
1312                         break;
1313                 case 'F':
1314                         /*
1315                          * Since we open the process read-only, there is no need
```

```
1316                         * for the -F flag.  It's a documented flag, so we
1317                         * consume it silently.
1318                         */
1319                        break;
1320                default:
1321                        errflg++;
1322                        break;
1323                }
1324        }

1326        /* -a is the default if no options are specified */
1327        if ((aflag + eflag + xflag + lflag) == 0) {
1328                aflag++;
1329                content |= CC_CONTENT_STACK;
1330        }

1332        /* -l cannot be used with the -x or -e flags */
1333        if (lflag && (xflag || eflag)) {
1334                (void) fprintf(stderr, "-l is incompatible with -x and -e\n");
1335                errflg++;
1336        }

1338        argc -= optind;
1339        argv += optind;

1341        if (errflg || argc <= 0) {
1342                (void) fprintf(stderr,
1343                    "usage:  %s [-aceFlx] { pid | core } ...\n"
1344                    "  (show process arguments and environment)\n"
1345                    "  -a: show process arguments (default)\n"
1346                    "  -c: interpret characters as 7-bit ascii regardless of "
1347                    "locale\n"
1348                    "  -e: show environment variables\n"
1349                    "  -F: force grabbing of the target process\n"
1350                    "  -l: display arguments as command line\n"
1351                    "  -x: show aux vector entries\n", command);
1352                return (2);
1353        }

1355        while (argc-- > 0) {
1356                char *arg;
1357                int gret, r;
1358                psinfo_t psinfo;
1359                char *psargs_conv;
1360                struct ps_prochandle *Pr;
1361                pargs_data_t datap;
1362                char *info;
1363                size_t info_sz;
1364                int pstate;
1365                char execname[PATH_MAX];
1366                int unprintable;
1367                int diflocale;

1369                (void) fflush(stdout);
1370                arg = *argv++;

1372                /*
1373                 * Suppress extra blanks lines if we've encountered processes
1374                 * which can't be opened.
1375                 */
1376                if (error == 0) {
1377                        (void) printf("\n");
1378                }
1379                error = 0;

1381                /*
```



```
1382                 * First grab just the psinfo information, in case this
1383                 * process is a zombie (in which case proc_arg_grab() will
1384                 * fail).  If so, print a nice message and continue.
1385                 */
1386                if (proc_arg_psinfo(arg, PR_ARG_ANY, &psinfo,
1387                    &gret) == -1) {
1388                        (void) fprintf(stderr, "%s: cannot examine %s: %s\n",
1389                            command, arg, Pgrab_error(gret));
1390                        retc++;
1391                        error = 1;
1392                        continue;
1393                }

1395                if (psinfo.pr_nlwp == 0) {
1396                        (void) printf("%d: <defunct>\n", (int)psinfo.pr_pid);
1397                        continue;
1398                }

1400                /*
1401                 * If process is a "system" process (like pageout), just
1402                 * print its psargs and continue on.
1403                 */
1404                if (psinfo.pr_size == 0 && psinfo.pr_rssize == 0) {
1405                        proc_unctrl_psinfo(&psinfo);
1406                        if (!lflag)
1407                                (void) printf("%d: ", (int)psinfo.pr_pid);
1408                        (void) printf("%s\n", psinfo.pr_psargs);
1409                        continue;
1410                }

1412                /*
1413                 * Open the process readonly, since we do not need to write to
1414                 * the control file.
1415                 */
1416                if ((Pr = proc_arg_grab(arg, PR_ARG_ANY, PGRAB_RDONLY,
1417                    &gret)) == NULL) {
1418                        (void) fprintf(stderr, "%s: cannot examine %s: %s\n",
1419                            command, arg, Pgrab_error(gret));
1420                        retc++;
1421                        error = 1;
1422                        continue;
1423                }

1425                pstate = Pstate(Pr);

1427                if (pstate == PS_DEAD &&
1428                    (Pcontent(Pr) & content) != content) {
1429                        (void) fprintf(stderr, "%s: core '%s' has "
1430                            "insufficient content\n", command, arg);
1431                        retc++;
1432                        continue;
1433                }

1435                /*
1436                 * If malloc() fails, we return here so that we can let go
1437                 * of the victim, restore our locale, print a message,
1438                 * then exit.
1439                 */
1440                if ((r = setjmp(env)) != 0) {
1441                        Prelease(Pr, 0);
1442                        (void) setlocale(LC_ALL, "");
1443                        (void) fprintf(stderr, "%s: out of memory: %s\n",
1444                            command, strerror(r));
1445                        return (1);
1446                }
```

```
1448                dmodel = Pstatus(Pr)->pr_dmodel;
1449                bzero(&datap, sizeof (datap));
1450                bcopy(Ppsinfo(Pr), &psinfo, sizeof (psinfo_t));
1451                datap.pd_proc = Pr;
1452                datap.pd_psinfo = &psinfo;

1454                if (cflag)
1455                        datap.pd_conv_flags |= CONV_STRICT_ASCII;

1457                /*
1458                 * Strip control characters, then record process summary in
1459                 * a buffer, since we don't want to print anything out until
1460                 * after we release the process.
1461                 */

1463                /*
1464                 * The process is neither a system process nor defunct.
1465                 *
1466                 * Do printing and post-processing (like name lookups) after
1467                 * gathering the raw data from the process and releasing it.
1468                 * This way, we don't deadlock on (for example) name lookup
1469                 * if we grabbed the nscd and do 'pargs -x'.
1470                 *
1471                 * We always fetch the environment of the target, so that we
1472                 * can make an educated guess about its locale.
1473                 */
1474                get_env(&datap);
1475                if (aflag != 0)
1476                        get_args(&datap);
1477                if (xflag != 0)
1478                        get_auxv(&datap);

1480                /*
1481                 * If malloc() fails after this poiint, we return here to
1482                 * restore our locale and print a message.  If we don't
1483                 * reset this, we might erroneously try to Prelease a process
1484                 * twice.
1485                 */
1486                if ((r = setjmp(env)) != 0) {
1487                        (void) setlocale(LC_ALL, "");
1488                        (void) fprintf(stderr, "%s: out of memory: %s\n",
1489                            command, strerror(r));
1490                        return (1);
1491                }

1493                /*
1494                 * For the -l option, we need a proper name for this executable
1495                 * before we release it.
1496                 */
1497                if (lflag)
1498                        datap.pd_execname = Pexecname(Pr, execname,
1499                            sizeof (execname));

1501                Prelease(Pr, 0);

1503                /*
1504                 * Crawl through the environment to determine the locale of
1505                 * the target.
1506                 */
1507                lookup_locale(&datap);
1508                diflocale = 0;
1509                setup_conversions(&datap, &diflocale);

1511                if (lflag != 0) {
1512                        unprintable = 0;
1513                        convert_array(&datap, datap.pd_argv_strs,
```



```
1514                            datap.pd_argc, &unprintable);
1515                        if (diflocale)
1516                                (void) fprintf(stderr, "%s: Warning, target "
1517                                    "locale differs from current locale\n",
1518                                    command);
1519                        else if (unprintable)
1520                                (void) fprintf(stderr, "%s: Warning, command "
1521                                    "line contains unprintable characters\n",
1522                                    command);

1524                        retc += print_cmdline(&datap);
1525                } else {
1526                        psargs_conv = convert_str(&datap, psinfo.pr_psargs,
1527                            &unprintable);
1528                        info_sz = strlen(psargs_conv) + MAXPATHLEN + 32 + 1;
1529                        info = malloc(info_sz);
1530                        if (pstate == PS_DEAD) {
1531                                (void) snprintf(info, info_sz,
1532                                    "core '%s' of %d:\t%s\n",
1533                                    arg, (int)psinfo.pr_pid, psargs_conv);
1534                        } else {
1535                                (void) snprintf(info, info_sz, "%d:\t%s\n",
1536                                    (int)psinfo.pr_pid, psargs_conv);
1537                        }
1538                        (void) printf("%s", info);
1539                        free(info);
1540                        free(psargs_conv);

1542                        if (aflag != 0) {
1543                                convert_array(&datap, datap.pd_argv_strs,
1544                                    datap.pd_argc, &unprintable);
1545                                print_args(&datap);
1546                                if (eflag || xflag)
1547                                        (void) printf("\n");
1548                        }

1550                        if (eflag != 0) {
1551                                convert_array(&datap, datap.pd_envp_strs,
1552                                    datap.pd_envc, &unprintable);
1553                                print_env(&datap);
1554                                if (xflag)
1555                                        (void) printf("\n");
1556                        }

1558                        if (xflag != 0) {
1559                                convert_array(&datap, datap.pd_auxv_strs,
1560                                    datap.pd_auxc, &unprintable);
1561                                print_auxv(&datap);
1562                        }
1563                }

1565                cleanup_conversions(&datap);
1566                free_data(&datap);
1567        }

1569        return (retc != 0 ? 1 : 0);
1570 }
```