```
**********************************************************
   24711 Mon Mar 31 18:53:08 2014
new/usr/src/cmd/xargs/xargs.c
4703 would like xargs support for -P
**********************************************************
```

```
/*
 * 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 2012 DEY Storage Systems, Inc.  All rights reserved.
 *
 * Portions of this file developed by DEY Storage Systems, Inc. are licensed
 * under the terms of the Common Development and Distribution License (CDDL)
 * version 1.0 only.  The use of subsequent versions of the License are
 * is specifically prohibited unless those terms are not in conflict with
 * version 1.0 of the License.  You can find this license on-line at
 * http://www.illumos.org/license/CDDL
 */
/*
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

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


#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdarg.h>
#include <stdlib.h>
#include <limits.h>
#include <wchar.h>
#include <locale.h>
#include <langinfo.h>
#include <stropts.h>
#include <poll.h>
#include <errno.h>
#include <stdarg.h>
#include "getresponse.h"

#define HEAD    0
#define TAIL    1
#define FALSE 0
#define TRUE 1
```



```
#define MAXSBUF 255
#define MAXIBUF 512
#define MAXINSERTS 5
#define BUFSIZE LINE_MAX
#define MAXARGS 255
#define INSPAT_STR      "{}"    /* default replstr string for -[Ii]     */
#define FORK_RETRY      5

#define QBUF_STARTLEN 255  /* start size of growable string buffer */
#define QBUF_INC 100       /* how much to grow a growable string by */

/* We use these macros to help make formatting look "consistent" */
#define EMSG(s)         ermsg(gettext(s "\n"))
#define EMSG2(s, a)     ermsg(gettext(s "\n"), a)
#define PERR(s)         perror(gettext("xargs: " s))

/* Some common error messages */

#define LIST2LONG       "Argument list too long"
#define ARG2LONG        "A single argument was greater than %d bytes"
#define MALLOCFAIL      "Memory allocation failure"
#define CORRUPTFILE     "Corrupt input file"
#define WAITFAIL        "Wait failure"
#define CHILDSIG        "Child killed with signal %d"
#define CHILDFAIL       "Command could not continue processing data"
#define FORKFAIL        "Could not fork child"
#define EXECFAIL        "Could not exec command"
#define MISSQUOTE       "Missing quote"
#define BADESCAPE       "Incomplete escape"
#define IBUFOVERFLOW    "Insert buffer overflow"
#define NOCHILDSLOT     "No free child slot available"
#endif /* ! codereview */

#define _(x)    gettext(x)

static wctype_t blank;
static char     *arglist[MAXARGS+1];
static char     argbuf[BUFSIZE * 2 + 1];
static char     lastarg[BUFSIZE + 1];
static char     **ARGV = arglist;
static char     *LEOF = "_";
static char     *INSPAT = INSPAT_STR;
static char     ins_buf[MAXIBUF];
static char     *p_ibuf;

static struct inserts {
        char    **p_ARGV;       /* where to put newarg ptr in arg list */
        char    *p_skel;        /* ptr to arg template */
} saveargv[MAXINSERTS];

static int      PROMPT = -1;
static int      BUFLIM = BUFSIZE;
static int      MAXPROCS = 1;
#endif /* ! codereview */
static int      N_ARGS = 0;
static int      N_args = 0;
static int      N_lines = 0;
static int      DASHX = FALSE;
static int      MORE = TRUE;
static int      PER_LINE = FALSE;
static int      ERR = FALSE;
static int      OK = TRUE;
static int      LEGAL = FALSE;
static int      TRACE = FALSE;
static int      INSERT = FALSE;
static int      ZERO = FALSE;
```

```
128 static int      linesize = 0;
129 static int      ibufsize = 0;
130 static int      exitstat = 0;   /* our exit status                      */
131 static int      mac;            /* modified argc, after parsing         */
132 static char     **mav;          /* modified argv, after parsing         */
133 static int      n_inserts;      /* # of insertions.                     */
134 static pid_t    *procs;         /* pids of children                     */
135 static int      n_procs;        /* # of child processes.                */
136 #endif /* ! codereview */

138 /* our usage message:                                                   */
139 #define USAGEMSG "Usage: xargs: [-t] [-p] [-0] [-e[eofstr]] [-E eofstr] "\
140         "[-I replstr] [-i[replstr]] [-L #] [-l[#]] [-n # [-x]] [-P maxprocs] [-s
 92         "[-I replstr] [-i[replstr]] [-L #] [-l[#]] [-n # [-x]] [-s size] "\
141         "[cmd [args ...]]\n"

143 static int      echoargs();
144 static wint_t   getwchr(char *, size_t *);
145 static void     lcall(char *sub, char **subargs);
 97 static int      lcall(char *sub, char **subargs);
146 static void     addibuf(struct inserts *p);
147 static void     ermsg(char *messages, ...);
148 static char     *addarg(char *arg);
149 static void     store_str(char **, char *, size_t);
150 static char     *getarg(char *);
151 static char     *insert(char *pattern, char *subst);
152 static void     usage();
153 static void     parseargs();
154 static void     procs_malloc(void);
155 static int      procs_find(pid_t child);
156 static void     procs_store(pid_t child);
157 static int      procs_delete(pid_t child);
158 static pid_t    procs_waitpid(int blocking, int *stat_loc);
159 static void     procs_wait(int blocking);
160 #endif /* ! codereview */

162 int
163 main(int argc, char **argv)
164 {
165         int     j;
166         struct inserts *psave;
167         int c;
168         int     initsize;
169         char    *cmdname, **initlist;
170         char    *arg;
171         char    *next;

173         /* initialization */
174         blank = wctype("blank");
175         n_inserts = 0;
176         psave = saveargv;
177         (void) setlocale(LC_ALL, "");
178 #if !defined(TEXT_DOMAIN)       /* Should be defined by cc -D           */
179 #define TEXT_DOMAIN "SYS_TEST"  /* Use this only if it weren't          */
180 #endif
181         (void) textdomain(TEXT_DOMAIN);
182         if (init_yes() < 0) {
183                 ermsg(_(ERR_MSG_INIT_YES), strerror(errno));
184                 exit(1);
185         }

187         parseargs(argc, argv);

189         /* handling all of xargs arguments:                             */
190         while ((c = getopt(mac, mav, "0tpe:E:I:i:L:l:n:P:s:x")) != EOF) {
106         while ((c = getopt(mac, mav, "0tpe:E:I:i:L:l:n:s:x")) != EOF) {
```



```
191                 switch (c) {
192                 case '0':
193                         ZERO = TRUE;
194                         break;

196                 case 't':       /* -t: turn trace mode on               */
197                         TRACE = TRUE;
198                         break;

200                 case 'p':       /* -p: turn on prompt mode.             */
201                         if ((PROMPT = open("/dev/tty", O_RDONLY)) == -1) {
202                                 PERR("can't read from tty for -p");
203                         } else {
204                                 TRACE = TRUE;
205                         }
206                         break;

208                 case 'e':
209                         /*
210                          * -e[eofstr]: set/disable end-of-file.
211                          * N.B. that an argument *isn't* required here; but
212                          * parseargs forced an argument if not was given.  The
213                          * forced argument is the default...
214                          */
215                         LEOF = optarg; /* can be empty */
216                         break;

218                 case 'E':
219                         /*
220                          * -E eofstr: change end-of-file string.
221                          * eofstr *is* required here, but can be empty:
222                          */
223                         LEOF = optarg;
224                         break;

226                 case 'I':
227                         /* -I replstr: Insert mode. replstr *is* required. */
228                         INSERT = PER_LINE = LEGAL = TRUE;
229                         N_ARGS = 0;
230                         INSPAT = optarg;
231                         if (*optarg == '\0') {
232                                 ermsg(_("Option requires an argument: -%c\n"),
233                                     c);
234                         }
235                         break;

237                 case 'i':
238                         /*
239                          * -i [replstr]: insert mode, with *optional* replstr.
240                          * N.B. that an argument *isn't* required here; if
241                          * it's not given, then the string INSPAT_STR will
242                          * be assumed.
243                          *
244                          * Since getopts(3C) doesn't handle the case of an
245                          * optional variable argument at all, we have to
246                          * parse this by hand:
247                          */

249                         INSERT = PER_LINE = LEGAL = TRUE;
250                         N_ARGS = 0;
251                         if ((optarg != NULL) && (*optarg != '\0')) {
252                                 INSPAT = optarg;
253                         } else {
254                                 /*
255                                  * here, there is no next argument. so
256                                  * we reset INSPAT to the INSPAT_STR.
```

```
257                                 * we *have* to do this, as -i/I may have
258                                 * been given previously, and XCU4 requires
259                                 * that only "the last one specified takes
260                                 * effect".
261                                 */
262                                INSPAT = INSPAT_STR;
263                        }
264                        break;

266                case 'L':
267                        /*
268                         * -L number: # of times cmd is executed
269                         * number *is* required here:
270                         */
271                        PER_LINE = TRUE;
272                        N_ARGS = 0;
273                        INSERT = FALSE;
274                        if ((PER_LINE = atoi(optarg)) <= 0) {
275                                ermsg(_("#lines must be positive int: %s\n"),
276                                    optarg);
277                        }
278                        break;

280                case 'l':
281                        /*
282                         * -l [number]: # of times cmd is executed
283                         * N.B. that an argument *isn't* required here; if
284                         * it's not given, then 1 is assumed.
285                         *
286                         * parseargs handles the optional arg processing.
287                         */

289                        PER_LINE = LEGAL = TRUE;  /* initialization     */
290                        N_ARGS = 0;
291                        INSERT = FALSE;

293                        if ((optarg != NULL) && (*optarg != '\0')) {
294                                if ((PER_LINE = atoi(optarg)) <= 0)
295                                        PER_LINE = 1;
296                        }
297                        break;

299                case 'n':       /* -n number: # stdin args              */
300                        /*
301                         * -n number: # stdin args.
302                         * number *is* required here:
303                         */
304                        if ((N_ARGS = atoi(optarg)) <= 0) {
305                                ermsg(_("#args must be positive int: %s\n"),
306                                    optarg);
307                        } else {
308                                LEGAL = DASHX || N_ARGS == 1;
309                                INSERT = PER_LINE = FALSE;
310                        }
311                        break;

313                case 'P':       /* -P maxprocs: # of child processses   */
314                        MAXPROCS = atoi(optarg);
315                        if (MAXPROCS <= 0) {
316                                ermsg(_("#maxprocs must be positive int: %s\n"),
317                                    optarg);
318                        }
319                        break;

321 #endif /* ! codereview */
322                case 's':       /* -s size: set max size of each arg list */
```



```
323                        BUFLIM = atoi(optarg);
324                        if (BUFLIM > BUFSIZE || BUFLIM <= 0) {
325                                ermsg(_("0 < max-cmd-line-size <= %d: %s\n"),
326                                    BUFSIZE, optarg);
327                        }
328                        break;

330                case 'x':       /* -x: terminate if args > size limit   */
331                        DASHX = LEGAL = TRUE;
332                        break;

334                default:
335                        /*
336                         * bad argument. complain and get ready to die.
337                         */
338                        usage();
339                        exit(2);
340                        break;
341                }
342        }

344        /*
345         * if anything called ermsg(), something screwed up, so
346         * we exit early.
347         */
348        if (OK == FALSE) {
349                usage();
350                exit(2);
351        }

353        /*
354         * we're finished handling xargs's options, so now pick up
355         * the command name (if any), and it's options.
356         */


359        mac -= optind;  /* dec arg count by what we've processed        */
360        mav += optind;  /* inc to current mav                           */

362        (void) procs_malloc();

364 #endif /* ! codereview */
365        if (mac <= 0) { /* if there're no more args to process, */
366                cmdname = "/usr/bin/echo";      /* our default command  */
367                *ARGV++ = addarg(cmdname);      /* use the default cmd. */
368        } else {        /* otherwise keep parsing rest of the string.   */
369                /*
370                 * note that we can't use getopts(3C), and *must* parse
371                 * this by hand, as we don't know apriori what options the
372                 * command will take.
373                 */
374                cmdname = *mav; /* get the command name */


377                /* pick up the remaining args from the command line:    */
378                while ((OK == TRUE) && (mac-- > 0)) {
379                        /*
380                         * while we haven't crapped out, and there's
381                         * work to do:
382                         */
383                        if (INSERT && ! ERR) {
384                                if (strstr(*mav, INSPAT) != NULL) {
385                                        if (++n_inserts > MAXINSERTS) {
386                                                ermsg(_("too many args "
387                                                    "with %s\n"), INSPAT);
388                                                ERR = TRUE;
```

```
389                                     }
390                                     psave->p_ARGV = ARGV;
391                                     (psave++)->p_skel = *mav;
392                             }
393                     }
394                     *ARGV++ = addarg(*mav++);
395             }
396     }

398     /* pick up args from standard input */

400     initlist = ARGV;
401     initsize = linesize;
402     lastarg[0] = '\0';

404     while (OK) {
405             N_args = 0;
406             N_lines = 0;
407             ARGV = initlist;
408             linesize = initsize;
409             next = argbuf;

411             while (MORE || (lastarg[0] != '\0')) {
412                     int l;

414                     if (*lastarg != '\0') {
415                             arg = strcpy(next, lastarg);
416                             *lastarg = '\0';
417                     } else if ((arg = getarg(next)) == NULL) {
418                             break;
419                     }

421                     l = strlen(arg) + 1;
422                     linesize += l;
423                     next += l;

425                     /* Inserts are handled specially later. */
426                     if ((n_inserts == 0) && (linesize >= BUFLIM)) {
427                             /*
428                              * Legal indicates hard fail if the list is
429                              * truncated due to size.  So fail, or if we
430                              * cannot create any list because it would be
431                              * too big.
432                              */
433                             if (LEGAL || N_args == 0) {
434                                     EMSG(LIST2LONG);
435                                     (void) procs_wait(1);
436 #endif /* ! codereview */
437                                     exit(2);
438                                     /* NOTREACHED */
439                             }

441                             /*
442                              * Otherwise just save argument for later.
443                              */
444                             (void) strcpy(lastarg, arg);
445                             break;
446                     }

448                     *ARGV++ = arg;

450                     N_args++;

452                     if ((PER_LINE && N_lines >= PER_LINE) ||
453                         (N_ARGS && (N_args) >= N_ARGS)) {
454                             break;
```



```
455                     }


458                     if ((ARGV - arglist) == MAXARGS) {
459                             break;
460                     }
461             }

463             *ARGV = NULL;
464             if (N_args == 0) {
465                     /* Reached the end with no more work. */
466                     break;
229                     exit(exitstat);
467             }

469             /* insert arg if requested */

471             if (!ERR && INSERT) {

473                     p_ibuf = ins_buf;
474                     ARGV--;
475                     j = ibufsize = 0;
476                     for (psave = saveargv; ++j <= n_inserts; ++psave) {
477                             addibuf(psave);
478                             if (ERR)
479                                     break;
480                     }
481             }
482             *ARGV = NULL;

484             if (n_inserts > 0) {
485                     /*
486                      * if we've done any insertions, re-calculate the
487                      * linesize. bomb out if we've exceeded our length.
488                      */
489                     linesize = 0;
490                     for (ARGV = arglist; *ARGV != NULL; ARGV++) {
491                             linesize += strlen(*ARGV) + 1;
492                     }
493                     if (linesize >= BUFLIM) {
494                             EMSG(LIST2LONG);
495                             (void) procs_wait(1);
496 #endif /* ! codereview */
497                             exit(2);
498                             /* NOTREACHED */
499                     }
500             }

502             /* exec command */

504             if (!ERR) {
505                     if (!MORE &&
506                         (PER_LINE && N_lines == 0 || N_ARGS && N_args == 0))
507                             exit(exitstat);
508                     OK = TRUE;
509                     j = TRACE ? echoargs() : TRUE;
510                     if (j) {
511                             /*
512                              * for xcu4, all invocations of cmdname must
513                              * return 0, in order for us to return 0.
514                              * so if we have a non-zero status here,
515                              * quit immediately.
516                              */
517                             (void) lcall(cmdname, arglist);
258                             exitstat |= lcall(cmdname, arglist);
518                     }
```

```
519                 }
520         }

522         (void) procs_wait(1);

524 #endif /* ! codereview */
525         if (OK)
526                 return (exitstat);

528         /*
529          * if exitstat was set, to match XCU4 complience,
530          * return that value, otherwise, return 1.
531          */
532         return (exitstat ? exitstat : 1);
533 }

535 static char *
536 addarg(char *arg)
537 {
538         linesize += (strlen(arg) + 1);
539         return (arg);
540 }


543 static void
544 store_str(char **buffer, char *str, size_t len)
545 {
546         (void) memcpy(*buffer, str, len);
547         (*buffer)[len] = '\0';
548         *buffer += len;
549 }


552 static char *
553 getarg(char *arg)
554 {
555         char    *xarg = arg;
556         wchar_t c;
557         char    mbc[MB_LEN_MAX];
558         size_t  len;
559         int     escape = 0;
560         int     inquote = 0;

562         arg[0] = '\0';

564         while (MORE) {

566                 len = 0;
567                 c = getwchr(mbc, &len);

569                 if (((arg - xarg) + len) > BUFLIM) {
570                         EMSG2(ARG2LONG, BUFLIM);
571                         exit(2);
572                         ERR = TRUE;
573                         return (NULL);
574                 }

576                 switch (c) {
577                 case '\n':
578                         if (ZERO) {
579                                 store_str(&arg, mbc, len);
580                                 continue;
581                         }
582                         /* FALLTHRU */

584                 case '\0':
```



```
585                 case WEOF:      /* Note WEOF == EOF */

587                         if (escape) {
588                                 EMSG(BADESCAPE);
589                                 ERR = TRUE;
590                                 return (NULL);
591                         }
592                         if (inquote) {
593                                 EMSG(MISSQUOTE);
594                                 ERR = TRUE;
595                                 return (NULL);
596                         }

598                         N_lines++;
599                         break;

601                 case '"':
602                         if (ZERO || escape || (inquote == 1)) {
603                                 /* treat it literally */
604                                 escape = 0;
605                                 store_str(&arg, mbc, len);

607                         } else if (inquote == 2) {
608                                 /* terminating double quote */
609                                 inquote = 0;

611                         } else {
612                                 /* starting quoted string */
613                                 inquote = 2;
614                         }
615                         continue;

617                 case '\'':
618                         if (ZERO || escape || (inquote == 2)) {
619                                 /* treat it literally */
620                                 escape = 0;
621                                 store_str(&arg, mbc, len);

623                         } else if (inquote == 1) {
624                                 /* terminating single quote */
625                                 inquote = 0;

627                         } else {
628                                 /* starting quoted string */
629                                 inquote = 1;
630                         }
631                         continue;

633                 case '\\':
634                         /*
635                          * Any unquoted character can be escaped by
636                          * preceding it with a backslash.
637                          */
638                         if (ZERO || inquote || escape) {
639                                 escape = 0;
640                                 store_str(&arg, mbc, len);
641                         } else {
642                                 escape = 1;
643                         }
644                         continue;

646                 default:
647                         /* most times we will just want to store it */
648                         if (inquote || escape || ZERO || !iswctype(c, blank)) {
649                                 escape = 0;
650                                 store_str(&arg, mbc, len);
```

```
651                                 continue;
652                         }
653                         /* unquoted blank */
654                         break;
655                 }

657                 /*
658                  * At this point we are processing a complete argument.
659                  */
660                 if (strcmp(xarg, LEOF) == 0 && *LEOF != '\0') {
661                         MORE = FALSE;
662                         return (NULL);
663                 }
664                 if (c == WEOF) {
665                         MORE = FALSE;
666                 }
667                 if (xarg[0] == '\0')
668                         continue;
669                 break;
670         }

672         return (xarg[0] == '\0' ? NULL : xarg);
673 }

675 /*
676  * ermsg():     print out an error message, and indicate failure globally.
677  *
678  *      Assumes that message has already been gettext()'d. It would be
679  *      nice if we could just do the gettext() here, but we can't, since
680  *      since xgettext(1M) wouldn't be able to pick up our error message.
681  */
682 /* PRINTFLIKE1 */
683 static void
684 ermsg(char *messages, ...)
685 {
686         va_list ap;

688         va_start(ap, messages);

690         (void) fprintf(stderr, "xargs: ");
691         (void) vfprintf(stderr, messages, ap);

693         va_end(ap);
694         OK = FALSE;
695 }

697 static int
698 echoargs()
699 {
700         char    **anarg;
701         char    **tanarg;       /* tmp ptr                      */
702         int             i;
703         char            reply[LINE_MAX];

705         tanarg = anarg = arglist-1;

707         /*
708          * write out each argument, separated by a space. the tanarg
709          * nonsense is for xcu4 testsuite compliance - so that an
710          * extra space isn't echoed after the last argument.
711          */
712         while (*++anarg) {              /* while there's an argument    */
713                 ++tanarg;               /* follow anarg                 */
714                 (void) write(2, *anarg, strlen(*anarg));

716                 if (*++tanarg) {        /* if there's another argument: */
```



```
717                         (void) write(2, " ", 1); /* add a space         */
718                         --tanarg;       /* reset back to anarg          */
719                 }
720         }
721         if (PROMPT == -1) {
722                 (void) write(2, "\n", 1);
723                 return (TRUE);
724         }

726         (void) write(2, "?...", 4);     /* ask the user for input       */

728         for (i = 0; i < LINE_MAX && read(PROMPT, &reply[i], 1) > 0; i++) {
729                 if (reply[i] == '\n') {
730                         if (i == 0)
731                                 return (FALSE);
732                         break;
733                 }
734         }
735         reply[i] = 0;

737         /* flush remainder of line if necessary */
738         if (i == LINE_MAX) {
739                 char    bitbucket;

741                 while ((read(PROMPT, &bitbucket, 1) > 0) && (bitbucket != '\n'))
742                         ;
743         }

745         return (yes_check(reply));
746 }


749 static char *
750 insert(char *pattern, char *subst)
751 {
752         static char     buffer[MAXSBUF+1];
753         int             len, ipatlen;
754         char    *pat;
755         char    *bufend;
756         char    *pbuf;

758         len = strlen(subst);
759         ipatlen = strlen(INSPAT) - 1;
760         pat = pattern - 1;
761         pbuf = buffer;
762         bufend = &buffer[MAXSBUF];

764         while (*++pat) {
765                 if (strncmp(pat, INSPAT, ipatlen) == 0) {
766                         if (pbuf + len >= bufend) {
767                                 break;
768                         } else {
769                                 (void) strcpy(pbuf, subst);
770                                 pat += ipatlen;
771                                 pbuf += len;
772                         }
773                 } else {
774                         *pbuf++ = *pat;
775                         if (pbuf >= bufend)
776                                 break;
777                 }
778         }

780         if (!*pat) {
781                 *pbuf = '\0';
782                 return (buffer);
```

```
783         } else {
784                 ermsg(gettext("Maximum argument size with insertion via %s's "
785                     "exceeded\n"), INSPAT);
786                 ERR = TRUE;
787                 return (NULL);
788         }
789 }


792 static void
793 addibuf(struct inserts  *p)
794 {
795         char    *newarg, *skel, *sub;
796         int             l;

798         skel = p->p_skel;
799         sub = *ARGV;
800         newarg = insert(skel, sub);
801         if (ERR)
802                 return;

804         l = strlen(newarg) + 1;
805         if ((ibufsize += l) > MAXIBUF) {
806                 EMSG(IBUFOVERFLOW);
807                 ERR = TRUE;
808         }
809         (void) strcpy(p_ibuf, newarg);
810         *(p->p_ARGV) = p_ibuf;
811         p_ibuf += l;
812 }


815 /*
816  * getwchr():   get the next wide character.
817  * description:
818  *      we get the next character from stdin.  This returns WEOF if no
819  *      character is present.  If ZERO is set, it gets a single byte instead
820  *      a wide character.
821  */
822 static wint_t
823 getwchr(char *mbc, size_t *sz)
824 {
825         size_t          i;
826         int             c;
827         wchar_t         wch;

829         i = 0;
830         while (i < MB_CUR_MAX) {

832                 if ((c = fgetc(stdin)) == EOF) {

834                         if (i == 0) {
835                                 /* TRUE EOF has been reached */
836                                 return (WEOF);
837                         }

839                         /*
840                          * We have some characters in our buffer still so it
841                          * must be an invalid character right before EOF.
842                          */
843                         break;
844                 }
845                 mbc[i++] = (char)c;

847                 /* If this succeeds then we are done */
848                 if (ZERO) {
```



```
849                         *sz = i;
850                         return ((char)c);
851                 }
852                 if (mbtowc(&wch, mbc, i) != -1) {
853                         *sz = i;
854                         return ((wint_t)wch);
855                 }
856         }

858         /*
859          * We have now encountered an illegal character sequence.
860          * There is nothing much we can do at this point but
861          * return an error.  If we attempt to recover we may in fact
862          * return garbage as arguments, from the customer's point
863          * of view.  After all what if they are feeding us a file
864          * generated in another locale?
865          */
866         errno = EILSEQ;
867         PERR(CORRUPTFILE);
868         exit(1);
869         /* NOTREACHED */
870 }


873 static void
263 static int
874 lcall(char *sub, char **subargs)
875 {
876         int     retry = 0;
877         pid_t   child;
266         int retcode, retry = 0;
267         pid_t iwait, child;

879         for (;;) {
880                 switch (child = fork()) {
881                 default:
882                         (void) procs_store(child);
883                         (void) procs_wait(0);
884                         return;
272                         while ((iwait = wait(&retcode)) != child &&
273                             iwait != (pid_t)-1)
274                                 ;
275                         if (iwait == (pid_t)-1) {
276                                 PERR(WAITFAIL);
277                                 exit(122);
278                                 /* NOTREACHED */
279                         }
280                         if (WIFSIGNALED(retcode)) {
281                                 EMSG2(CHILDSIG, WTERMSIG(retcode));
282                                 exit(125);
283                                 /* NOTREACHED */
284                         }
285                         if ((WEXITSTATUS(retcode) & 0377) == 0377) {
286                                 EMSG(CHILDFAIL);
287                                 exit(124);
288                                 /* NOTREACHED */
289                         }
290                         return (WEXITSTATUS(retcode));
885                 case 0:
886                         (void) execvp(sub, subargs);
887                         PERR(EXECFAIL);
888                         if (errno == EACCES)
889                                 exit(126);
890                         exit(127);
891                         /* NOTREACHED */
892                 case -1:
```

```
                        if (errno != EAGAIN && retry++ < FORK_RETRY) {
                                PERR(FORKFAIL);
                                exit(123);
                        }
                        (void) sleep(1);
                }
        }
}

static void
procs_malloc(void)
{
        int     i;

        procs = (pid_t *)(malloc(MAXPROCS * sizeof(pid_t)));
        if (procs == NULL) {
                PERR(MALLOCFAIL);
                exit(1);
        }

        for (i = 0; i < MAXPROCS; i++) {
                procs[i] = (pid_t)(0);
        }
}

static int
procs_find(pid_t child)
{
        int     i;

        for (i = 0; i < MAXPROCS; i++) {
                if (procs[i] == child) {
                        return (i);
                }
        }

        return (-1);
}

static void
procs_store(pid_t child)
{
        int     i;

        i = procs_find((pid_t)(0));
        if (i < 0) {
                PERR(NOCHILDSLOT);
                exit(1);
        }
        procs[i] = child;
        n_procs++;
}

static int
procs_delete(pid_t child)
{
        int     i;

        i = procs_find(child);
        if (i < 0) {
                return (0);
        }
        procs[i] = (pid_t)(0);
        n_procs--;
        return (1);
}
```



```
static pid_t
procs_waitpid(int blocking, int *stat_loc)
{
        pid_t   child;
        int     options;

        if (n_procs == 0) {
                errno = ECHILD;
                return (-1);
        }

        options = (blocking) ? 0 : WNOHANG;

        while ((child = waitpid(-1, stat_loc, options)) > 0) {
                if (procs_delete(child)) {
                        break;
                }
        }

        return (child);
}

static void
procs_wait(int blocking)
{
        pid_t   child;
        int     stat_loc;

        while ((child = procs_waitpid(blocking || (n_procs >= MAXPROCS) ? 1 : 0,
                if (WIFSIGNALED(stat_loc)) {
                        EMSG2(CHILDSIG, WTERMSIG(stat_loc));
                        exit(125);
                        /* NOTREACHED */
                } else if ((WEXITSTATUS(stat_loc) & 0377) == 0377) {
                        EMSG(CHILDFAIL);
                        exit(124);
                        /* NOTREACHED */
                } else {
                        exitstat |= WEXITSTATUS(stat_loc);
                }
        }

        if (child == (pid_t)(-1) && errno != ECHILD) {
                EMSG(WAITFAIL);
                exit(122);
                /* NOTREACHED */
        }
}
#endif /* ! codereview */

static void
usage()
{
        ermsg(_(USAGEMSG));
        OK = FALSE;
}



/*
 * parseargs():         modify the args
 *      since the -e, -i and -l flags all take optional subarguments,
 *      and getopts(3C) is clueless about this nonsense, we change the
 *      our local argument count and strings to separate this out,
 *      and make it easier to handle via getopts(3c).
```

```
 *
 *      -e      -> "-e ""
 *      -e3     -> "-e "3"
 *      -Estr   -> "-E "str"
 *      -i      -> "-i "{}"
 *      -irep   -> "-i "rep"
 *      -l      -> "-i "1"
 *      -l10    -> "-i "10"
 *
 *      since the -e, -i and -l flags all take optional subarguments,
 */
static void
parseargs(int ac, char **av)
{
        int i;                  /* current argument                     */
        int cflag;              /* 0 = not processing cmd arg           */

        if ((mav = malloc((ac * 2 + 1) * sizeof (char *))) == NULL) {
                PERR(MALLOCFAIL);
                exit(1);
        }

        /* for each argument, see if we need to change things:          */
        for (i = mac = cflag = 0; (av[i] != NULL) && i < ac; i++, mac++) {
                if ((mav[mac] = strdup(av[i])) == NULL) {
                        PERR(MALLOCFAIL);
                        exit(1);
                }

                /* -- has been found or argument list is fully processes */
                if (cflag)
                        continue;

                /*
                 * if we're doing special processing, and we've got a flag
                 */
                else if ((av[i][0] == '-') && (av[i][1] != NULL)) {
                        char    *def;

                        switch (av[i][1]) {
                        case    'e':
                                def = ""; /* -e with no arg turns off eof */
                                goto process_special;
                        case    'i':
                                def = INSPAT_STR;
                                goto process_special;
                        case    'l':
                                def = "1";
process_special:
                                /*
                                 * if there's no sub-option, we *must* add
                                 * a default one. this is because xargs must
                                 * be able to distinguish between a valid
                                 * suboption, and a command name.
                                 */
                                if (av[i][2] == NULL) {
                                        mav[++mac] = strdup(def);
                                } else {
                                        /* clear out our version: */
                                        mav[mac][2] = NULL;
                                        mav[++mac] = strdup(&av[i][2]);
                                }
                                if (mav[mac] == NULL) {
                                        PERR(MALLOCFAIL);
                                        exit(1);
                                }
```



```
                                break;

                        /* flags with required subarguments:            */

                        /*
                         * there are two separate cases here. either the
                         * flag can have the normal XCU4 handling
                         * (of the form: -X subargument); or it can have
                         * the old solaris 2.[0-4] handling (of the
                         * form: -Xsubargument). in order to maintain
                         * backwards compatibility, we must support the
                         * latter case. we handle the latter possibility
                         * first so both the old solaris way of handling
                         * and the new XCU4 way of handling things are allowed.
                         */
                        case    'n':    /* FALLTHROUGH                  */
                        case    'P':    /* FALLTHROUGH                  */
#endif /* ! codereview */
                        case    's':    /* FALLTHROUGH                  */
                        case    'E':    /* FALLTHROUGH                  */
                        case    'I':    /* FALLTHROUGH                  */
                        case    'L':
                                /*
                                 * if the second character isn't null, then
                                 * the user has specified the old syntax.
                                 * we move the subargument into our
                                 * mod'd argument list.
                                 */
                                if (av[i][2] != NULL) {
                                        /* first clean things up:       */
                                        mav[mac][2] = NULL;

                                        /* now add the separation:      */
                                        ++mac;  /* inc to next mod'd arg */
                                        if ((mav[mac] = strdup(&av[i][2])) ==
                                            NULL) {
                                                PERR(MALLOCFAIL);
                                                exit(1);
                                        }
                                        break;
                                }
                                i++;
                                mac++;

                                if (av[i] == NULL) {
                                        mav[mac] = NULL;
                                        return;
                                }
                                if ((mav[mac] = strdup(av[i])) == NULL) {
                                        PERR(MALLOCFAIL);
                                        exit(1);
                                }
                                break;

                        /* flags */
                        case 'p' :
                        case 't' :
                        case 'x' :
                        case '0' :
                                break;

                        case '-' :
                        default:
                                /*
                                 * here we've hit the cmd argument. so
                                 * we'll stop special processing, as the
```

```
1157                                 * cmd may have a "-i" etc., argument,
1158                                 * and we don't want to add a "" to it.
1159                                 */
1160                                cflag = 1;
1161                                break;
1162                        }
1163                } else if (i > 0) {     /* if we're not the 1st arg     */
1164                        /*
1165                         * if it's not a flag, then it *must* be the cmd.
1166                         * set cflag, so we don't mishandle the -[eil] flags.
1167                         */
1168                        cflag = 1;
1169                }
1170        }

1172        mav[mac] = NULL;
1173 }
```

```
**********************************************************
   15706 Mon Mar 31 18:53:09 2014
new/usr/src/man/man1/xargs.1
4703 would like xargs support for -P
**********************************************************
```

```
   1 '\" te
   2 .\"  Copyright 1989 AT&T Copyright (c) 1992, X/Open Company Limited All Rights R
   3 .\" Sun Microsystems, Inc. gratefully acknowledges The Open Group for permission
   4 .\" http://www.opengroup.org/bookstore/.
   5 .\" The Institute of Electrical and Electronics Engineers and The Open Group, ha
   6 .\"  This notice shall appear on any product containing this material.
   7 .\" The contents of this file are subject to the terms of the Common Development
   8 .\" You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE or http:
   9 .\" When distributing Covered Code, include this CDDL HEADER in each file and in
  10 .TH XARGS 1 "March 31, 2014"
  10 .TH XARGS 1 "November 24, 2012"
  11 .SH NAME
  12 xargs \- construct argument lists and invoke utility
  13 .SH SYNOPSIS
  14 .LP
  15 .nf
  16 \fBxargs\fR [\fB-t\fR] [\fB-0\fR] [\fB-p\fR] [\fB-e\fR[\fIeofstr\fR]] [\fB-E\fR
  17      [\fB-I\fR \fIreplstr\fR] [\fB-i\fR[\fIreplstr\fR]] [\fB-L\fR \fInumber\fR]
  18      [\fB-n\fR \fInumber\fR [\fB-x\fR]] [\fB-P\fR \fImaxprocs\fR] [\fB-s\fR \fIs
  19      [\fIutility\fR [\fIargument\fR...]]
  18      [\fB-n\fR \fInumber\fR [\fB-x\fR]] [\fB-s\fR \fIsize\fR] [\fIutility\fR [\f
  20 .fi

  22 .SH DESCRIPTION
  23 .sp
  24 .LP
  25 The \fBxargs\fR utility constructs a command line consisting of the
  26 \fIutility\fR and \fIargument\fR operands specified followed by as many
  27 arguments read in sequence from standard input as fit in length and number
  28 constraints specified by the options. The \fBxargs\fR utility then invokes the
  29 constructed command line and waits for its completion. This sequence is
  30 repeated until an end-of-file condition is detected on standard input or an
  31 invocation of a constructed command line returns an exit status of \fB255\fR.
  32 .sp
  33 .LP
  34 Arguments in the standard input must be separated by unquoted blank characters,
  35 or unescaped blank characters or newline characters. A string of zero or more
  36 non-double-quote (\fB"\fR) and non-newline characters can be quoted by
  37 enclosing them in double-quotes. A string of zero or more non-apostrophe
  38 (\fB\&'\fR) and non-newline characters can be quoted by enclosing them in
  39 apostrophes. Any unquoted character can be escaped by preceding it with a
  40 backslash (\fB\e\fR). The \fIutility\fR are executed one or more times until
  41 the end-of-file is reached. The results are unspecified if the utility named by
  42 \fIutility\fR attempts to read from its standard input.
  43 .sp
  44 .LP
  45 The generated command line length is the sum of the size in bytes of the
  46 utility name and each argument treated as strings, including a null byte
  47 terminator for each of these strings. The \fBxargs\fR utility limits the
  48 command line length such that when the command line is invoked, the combined
  49 argument and environment lists can not exceed \fB{ARG_MAX}\(mi2048\fR bytes.
  50 Within this constraint, if neither the \fB-n\fR nor the \fB-s\fR option is
  51 specified, the default command line length is at least \fB{LINE_MAX}\fR.
  52 .SH OPTIONS
  53 .sp
  54 .LP
  55 The following options are supported:
  56 .sp
  57 .ne 2
  58 .na
  59 \fB\fB\fR\fB-e\fR\fB[\fR\fIeofstr\fR\fB]\fR\fR
```



```
  60 .ad
  61 .RS 15n
  62 Uses \fIeofstr\fR as the logical end-of-file string. Underscore (\fB_\fR) is
  63 assumed for the logical \fBEOF\fR string if neither \fB-e\fR nor \fB-E\fR is
  64 used. When the \fIeofstr\fR option-argument is omitted, the logical \fBEOF\fR
  65 string capability is disabled and underscores are taken literally. The
  66 \fBxargs\fR utility reads standard input until either end-of-file or the
  67 logical \fBEOF\fR string is encountered.
  68 .RE

  70 .sp
  71 .ne 2
  72 .na
  73 \fB\fB-E\fR \fIeofstr\fR\fR
  74 .ad
  75 .RS 15n
  76 Specifies a logical end-of-file string to replace the default underscore.
  77 \fBxargs\fR reads standard input until either end-of-file or the logical EOF
  78 string is encountered. When \fIeofstr\fR is a null string, the logical
  79 end-of-file string capability is disabled and underscore characters are taken
  80 literally.
  81 .RE

  83 .sp
  84 .ne 2
  85 .na
  86 \fB\fB-I\fR \fIreplstr\fR\fR
  87 .ad
  88 .RS 15n
  89 Insert mode. \fIutility\fR is executed for each line from standard input,
  90 taking the entire line as a single argument, inserting it in \fIargument\fR
  91 \fIs\fR for each occurrence of \fIreplstr\fR. A maximum of five arguments in
  92 \fIargument\fRs can each contain one or more instances of \fIreplstr\fR. Any
  93 blank characters at the beginning of each line are ignored. Constructed
  94 arguments cannot grow larger than 255 bytes. Option \fB-x\fR is forced on. The
  95 \fB-I\fR and \fB-i\fR options are mutually exclusive; the last one specified
  96 takes effect.
  97 .RE

  99 .sp
 100 .ne 2
 101 .na
 102 \fB\fB\fR\fB-i\fR\fB[\fR\fIreplstr\fR\fB]\fR\fR
 103 .ad
 104 .RS 15n
 105 This option is equivalent to \fB-I\fR \fIreplstr\fR. The string \fB{\|}\fR is
 106 assumed for \fIreplstr\fR if the option-argument is omitted.
 107 .RE

 109 .sp
 110 .ne 2
 111 .na
 112 \fB\fB-L\fR \fInumber\fR\fR
 113 .ad
 114 .RS 15n
 115 The \fIutility\fR is executed for each non-empty \fInumber\fR lines of
 116 arguments from standard input. The last invocation of \fIutility\fR is with
 117 fewer lines of arguments if fewer than \fInumber\fR remain. A line is
 118 considered to end with the first newline character unless the last character of
 119 the line is a blank character; a trailing blank character signals continuation
 120 to the next non-empty line, inclusive. The \fB-L\fR, \fB-l\fR, and \fB-n\fR
 121 options are mutually exclusive; the last one specified takes effect.
 122 .RE

 124 .sp
 125 .ne 2
```

```
.na
\fB\fB-l[\fR\fInumber\fR\fB]\fR\fR
.ad
.RS 15n
(The letter ell.) This option is equivalent to \fB-L\fR \fInumber\fR. If
\fInumber\fR is omitted, \fB1\fR is assumed. Option \fB-x\fR is forced on.
.RE

.sp
.ne 2
.na
\fB\fB-n\fR \fInumber\fR\fR
.ad
.RS 15n
Invokes \fIutility\fR using as many standard input arguments as possible, up to
\fInumber\fR (a positive decimal integer) arguments maximum. Fewer arguments
are used if:
.RS +4
.TP
.ie t \(bu
.el o
The command line length accumulated exceeds the size specified by the \fB-s\fR
option (or \fB{LINE_MAX}\fR if there is no \fB-s\fR option), or
.RE
.RS +4
.TP
.ie t \(bu
.el o
The last iteration has fewer than \fInumber\fR, but not zero, operands
remaining.
.RE
.RE

.sp
.ne 2
.na
\fB-p\fR
.ad
.RS 15n
Prompt mode. The user is asked whether to execute \fIutility\fR at each
invocation. Trace mode (\fB-t\fR) is turned on to write the command instance to
be executed, followed by a prompt to standard error. An affirmative response
(specific to the user's locale) read from \fB/dev/tty\fR executes the command;
otherwise, that particular invocation of \fIutility\fR is skipped.
.RE

.sp
.ne 2
.na
\fB\fB-P\fR \fImaxprocs\fR\fR
.ad
.RS 15n
Invokes \fIutility\fR using at most \fImaxprocs\fR (a positive decimal integer)
parallel child processes.
#endif /* ! codereview */
.RE

.sp
.ne 2
.na
\fB\fB-s\fR \fIsize\fR\fR
.ad
.RS 15n
Invokes \fIutility\fR using as many standard input arguments as possible
yielding a command line length less than \fIsize\fR (a positive decimal
integer) bytes. Fewer arguments are used if:
```



```
.RS +4
.TP
.ie t \(bu
.el o
The total number of arguments exceeds that specified by the \fB-n\fR option, or
.RE
.RS +4
.TP
.ie t \(bu
.el o
The total number of lines exceeds that specified by the \fB-L\fR option, or
.RE
.RS +4
.TP
.ie t \(bu
.el o
End of file is encountered on standard input before \fIsize\fR bytes are
accumulated.
.RE
Values of \fIsize\fR up to at least \fB{LINE_MAX}\fR bytes are supported,
provided that the constraints specified in DESCRIPTION are met. It is not
considered an error if a value larger than that supported by the implementation
or exceeding the constraints specified in DESCRIPTION is specified. \fBxargs\fR
uses the largest value it supports within the constraints.
.RE

.sp
.ne 2
.na
\fB\fB-t\fR\fR
.ad
.RS 6n
Enables trace mode. Each generated command line is written to standard error
just prior to invocation.
.RE

.sp
.ne 2
.na
\fB\fB-x\fR\fR
.ad
.RS 6n
Terminates if a command line containing \fInumber\fR arguments (see the
\fB-n\fR option above) or \fInumber\fR lines (see the \fB-L\fR option above)
does not fit in the implied or specified size (see the \fB-s\fR option above).
.RE

.sp
.ne 2
.na
\fB-0\fR
.ad
.RS 6n
Null separator mode.  Instead of using white space or new lines to
delimit arguments, zero bytes are used.  This is suitable for use with
the -print0 argument to \fBfind\fR(1).
.RE

.SH OPERANDS
.sp
.LP
The following operands are supported:
.sp
.ne 2
.na
\fB\fIutility\fR\fR
```

```
.ad
.RS 12n
The name of the utility to be invoked, found by search path using the
\fBPATH\fR environment variable. (ee \fBenviron\fR(5).) If \fIutility\fR is
omitted, the default is the \fBecho\fR(1) utility. If the \fIutility\fR operand
names any of the special built-in utilities in \fBshell_builtins\fR(1), the
results are undefined.
.RE

.sp
.ne 2
.na
\fB\fIargument\fR\fR
.ad
.RS 12n
An initial option or operand for the invocation of \fIutility\fR.
.RE

.SH USAGE
.sp
.LP
The \fB255\fR exit status allows a utility being used by \fBxargs\fR to tell
\fBxargs\fR to terminate if it knows no further invocations using the current
data stream succeeds. Thus, \fIutility\fR should explicitly \fBexit\fR with an
appropriate value to avoid accidentally returning with \fB255\fR.
.sp
.LP
Notice that input is parsed as lines. Blank characters separate arguments. If
\fBxargs\fR is used to bundle output of commands like \fBfind\fR \fIdir\fR
\fB-print\fR or \fBls\fR into commands to be executed, unexpected results are
likely if any filenames contain any blank characters or newline characters.
This can be fixed by using \fBfind\fR to call a script that converts each file
found into a quoted string that is then piped to \fBxargs\fR. Notice that the
quoting rules used by \fBxargs\fR are not the same as in the shell. They were
not made consistent here because existing applications depend on the current
rules and the shell syntax is not fully compatible with it. An easy rule that
can be used to transform any string into a quoted form that \fBxargs\fR
interprets correctly is to precede each character in the string with a
backslash (\fB\e\fR).
.sp
.LP
On implementations with a large value for \fB{ARG_MAX}\fR, \fBxargs\fR can
produce command lines longer than \fB{LINE_MAX}\fR. For invocation of
utilities, this is not a problem. If \fBxargs\fR is being used to create a text
file, users should explicitly set the maximum command line length with the
\fB-s\fR option.
.sp
.LP
The \fBxargs\fR utility returns exit status \fB127\fR if an error occurs so
that applications can distinguish "failure to find a utility" from "invoked
utility exited with an error indication." The value \fB127\fR was chosen
because it is not commonly used for other meanings; most utilities use small
values for "normal error conditions" and the values above \fB128\fR can be
confused with termination due to receipt of a signal. The value \fB126\fR was
chosen in a similar manner to indicate that the utility could be found, but not
invoked.
.SH EXAMPLES
.LP
\fBExample 1 \fRUsing the xargs command
.sp
.LP
The following example moves all files from directory \fB$1\fR to directory
\fB$2\fR, and echo each move command just before doing it:

.sp
.in +2
```



```
.nf
example% \fBls $1 | xargs -I {} -t mv $1/{} $2/{}\fR
.fi
.in -2
.sp

.sp
.LP
The following command combines the output of the parenthesised commands onto
one line, which is then written to the end of file \fBlog\fR:

.sp
.in +2
.nf
example% \fB(logname; date; printf "%s\en" "$0 $*") | xargs >>log\fR
.fi
.in -2
.sp

.sp
.LP
The following command invokes \fBdiff\fR with successive pairs of arguments
originally typed as command line arguments (assuming there are no embedded
blank characters in the elements of the original argument list):

.sp
.in +2
.nf
example% \fBprintf "%s\en" "$*" | xargs -n 2 -x diff\fR
.fi
.in -2
.sp

.sp
.LP
The user is asked which files in the current directory are to be archived. The
files are archived into \fBarch\fR \fB;\fR a, one at a time, or b, many at a
time:

.sp
.in +2
.nf
example% \fBls | xargs -p -L 1 ar -r arch
ls | xargs -p -L 1 | xargs ar -r arch\fR
.fi
.in -2
.sp

.sp
.LP
The following executes with successive pairs of arguments originally typed as
command line arguments:

.sp
.in +2
.nf
example% \fBecho $* | xargs -n 2 diff\fR
.fi
.in -2
.sp

.SH ENVIRONMENT VARIABLES
.sp
.LP
See \fBenviron\fR(5) for descriptions of the following environment variables
that affect the execution of \fBxargs\fR: \fBLANG\fR, \fBLC_ALL\fR,
```

```
\fBLC_COLLATE\fR, \fBLC_CTYPE\fR, \fBLC_MESSAGES\fR, and \fBNLSPATH\fR.
.sp
.ne 2
.na
\fB\fBPATH\fR\fR
.ad
.RS 8n
Determine the location of \fIutility\fR.
.RE

.sp
.LP
Affirmative responses are processed using the extended regular expression
defined for the \fByesexpr\fR keyword in the \fBLC_MESSAGES\fR category of the
user's locale. The locale specified in the \fBLC_COLLATE\fR category defines
the behavior of ranges, equivalence classes, and multi-character collating
elements used in the expression defined for \fByesexpr\fR. The locale specified
in \fBLC_CTYPE\fR determines the locale for interpretation of sequences of
bytes of text data a characters, the behavior of character classes used in the
expression defined for the \fByesexpr\fR. See \fBlocale\fR(5).
.SH EXIT STATUS
.sp
.LP
The following exit values are returned:
.sp
.ne 2
.na
\fB\fB0\fR\fR
.ad
.RS 12n
All invocations of \fIutility\fR returned exit status \fB0\fR.
.RE

.sp
.ne 2
.na
\fB\fB1\(mi125\fR\fR
.ad
.RS 12n
A command line meeting the specified requirements could not be assembled, one
or more of the invocations of \fIutility\fR returned a non-zero exit status, or
some other error occurred.
.RE

.sp
.ne 2
.na
\fB\fB126\fR\fR
.ad
.RS 12n
The utility specified by \fIutility\fR was found but could not be invoked.
.RE

.sp
.ne 2
.na
\fB\fB127\fR\fR
.ad
.RS 12n
The utility specified by \fIutility\fR could not be found.
.RE

.sp
.LP
If a command line meeting the specified requirements cannot be assembled, the
utility cannot be invoked, an invocation of the utility is terminated by a
```



```
signal, or an invocation of the utility exits with exit status \fB255\fR, the
\fBxargs\fR utility writes a diagnostic message and exit without processing any
remaining input.
.SH ATTRIBUTES
.sp
.LP
See \fBattributes\fR(5) for descriptions of the following attributes:
.sp

.sp
.TS
box;
c | c
l | l .
ATTRIBUTE TYPE  ATTRIBUTE VALUE
_
CSI     Enabled
_
Interface Stability     Standard
.TE

.SH SEE ALSO
.sp
.LP
\fBecho\fR(1), \fBshell_builtins\fR(1), \fBattributes\fR(5), \fBenviron\fR(5),
\fBstandards\fR(5)
```