

new/usr/src/cmd/xargs/xargs.c

```
*****
24711 Mon Mar 31 18:53:08 2014
new/usr/src/cmd/xargs/xargs.c
4703 would like xargs support for -P
*****
1 /* CDDL HEADER START
2 *
3 * The contents of this file are subject to the terms of the
4 * Common Development and Distribution License (the "License").
5 * You may not use this file except in compliance with the License.
6 *
7 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
8 * or http://www.opensolaris.org/os/licensing.
9 * See the License for the specific language governing permissions
10 * and limitations under the License.
11 *
12 * When distributing Covered Code, include this CDDL HEADER in each
13 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
14 * If applicable, add the following below this CDDL HEADER, with the
15 * fields enclosed by brackets "[]" replaced with your own identifying
16 * information: Portions Copyright [yyyy] [name of copyright owner]
17 *
18 * CDDL HEADER END
19 */
20 /*
21 * Copyright 2012 DEY Storage Systems, Inc. All rights reserved.
22 *
23 * Portions of this file developed by DEY Storage Systems, Inc. are licensed
24 * under the terms of the Common Development and Distribution License (CDDL)
25 * version 1.0 only. The use of subsequent versions of the License are
26 * specifically prohibited unless those terms are not in conflict with
27 * version 1.0 of the License. You can find this license on-line at
28 * http://www.illumos.org/license/CDDL
29 */
30 /*
31 * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
32 * Use is subject to license terms.
33 */
34 */

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

37 /*
40 #include <stdio.h>
41 #include <sys/types.h>
42 #include <sys/wait.h>
43 #include <unistd.h>
44 #include <fcntl.h>
45 #include <string.h>
46 #include <stdarg.h>
47 #include <stdlib.h>
48 #include <limits.h>
49 #include <wchar.h>
50 #include <locale.h>
51 #include <langinfo.h>
52 #include <stropts.h>
53 #include <poll.h>
54 #include <errno.h>
55 #include <stdarg.h>
56 #include "getresponse.h"

58 #define HEAD 0
59 #define TAIL 1
60 #define FALSE 0
61 #define TRUE 1
```

1

new/usr/src/cmd/xargs/xargs.c

```
62 #define MAXSBUF 255
63 #define MAXIBUF 512
64 #define MAXINSERTS 5
65 #define BUFSIZE LINE_MAX
66 #define MAXARGS 255
67 #define INSPAT_STR      "{}"    /* default replstr string for -[Ii] */
68 #define FORK_RETRY      5

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

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

78 /* Some common error messages */

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

95 #define _(x)    gettext(x)

97 static wctype_t blank;
98 static char   *arglist[MAXARGS+1];
99 static char   argbuf[BUFSIZE * 2 + 1];
100 static char  lastarg[BUFSIZE + 1];
101 static char  **ARGV = arglist;
102 static char  *LEOF = "_";
103 static char  *INSPAT = INSPAT_STR;
104 static char  ins_buf[MAXIBUF];
105 static char  *p_ibuf;

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

112 static int    PROMPT = -1;
113 static int    BUFLIM = BUFSIZE;
114 static int    MAXPROCS = 1;
115 #endif /* ! codereview */
116 static int    N_ARGS = 0;
117 static int    N_args = 0;
118 static int    N_lines = 0;
119 static int    DASHX = FALSE;
120 static int    MORE = TRUE;
121 static int    PER_LINE = FALSE;
122 static int    ERR = FALSE;
123 static int    OK = TRUE;
124 static int    LEGAL = FALSE;
125 static int    TRACE = FALSE;
126 static int    INSERT = FALSE;
127 static int    ZERO = FALSE;
```

2

```

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);
146 static int    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, "Otp:E:I:i:L:l:n:P:s:x")) != EOF) {
191         while ((c = getopt(mac, mav, "Otp: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;
199             case 'p': /* -p: turn on prompt mode. */
200                 if ((PROMPT = open("/dev/tty", O_RDONLY)) == -1) {
201                     PERR("can't read from tty for -p");
202                 } else {
203                     TRACE = TRUE;
204                 }
205                 break;
206             case 'e':
207                 /* -e[eofstr]: set/disable end-of-file.
208                  * N.B. that an argument *isn't* required here; but
209                  * parseargs forced an argument if not was given. The
210                  * forced argument is the default...
211                 */
212                 LEOF = optarg; /* can be empty */
213                 break;
214             case 'E':
215                 /* -E eofstr: change end-of-file string.
216                  * eofstr *is* required here, but can be empty:
217                 */
218                 LEOF = optarg;
219                 break;
220             case 'I':
221                 /* -I replstr: Insert mode. replstr *is* required. */
222                 INSERT = PER_LINE = LEGAL = TRUE;
223                 N_ARGS = 0;
224                 INSPAT = optarg;
225                 if (*optarg == '\0') {
226                     ermsg(_("Option requires an argument: -%c\n"),
227                           c);
227                 }
228                 break;
229             case 'i':
230                 /* -i [replstr]: insert mode, with *optional* replstr.
231                  * N.B. that an argument *isn't* required here; if
232                  * it's not given, then the string INSPAT_STR will
233                  * be assumed.
234                  *
235                  * Since getopt(3C) doesn't handle the case of an
236                  * optional variable argument at all, we have to
237                  * parse this by hand:
238                 */
239                 INSERT = PER_LINE = LEGAL = TRUE;
240                 N_ARGS = 0;
241                 if ((optarg != NULL) && (*optarg != '\0')) {
242                     INSPAT = optarg;
243                 } else {
244                     /*
245                      * here, there is no next argument. so
246                      * we reset INSPAT to the INSPAT_STR.
247                 */

```

new/usr/src/cmd/xargs/xargs.c

四

```

new/usr/src/cmd/xargs/xargs.c

323             BUFLIM = atoi(optarg);
324             if (BUFLIM > BUFSIZE || BUFLIM <= 0) {
325                     errmsg(_("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 errmsg(), 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  */
357

359     mac -= optind; /* dec arg count by what we've processed
360     mav += optind; /* inc to current mav */
361

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 getopt(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 */

375     /* pick up the remaining args from the command line: */
376     while ((OK == TRUE) && (mac-- > 0)) {
377         /*
378          * while we haven't crapped out, and there's
379          * work to do:
380          */
381         if (INSERT && ! ERR) {
382             if (strstr(*mav, INSPAT) != NULL) {
383                 if (++n_inserts > MAXINSERTS) {
384                     errmsg(("too many args "
385                           "with %s\n"), INSPAT);
386                     ERR = TRUE;
387                 }
388             }
389         }
390     }
391 }

393     if (ERR) {
394         if (errno)
395             perror("xargs");
396         exit(1);
397     }
398 }

399     if (mav)
400         free(mav);
401 }

402     if (OK == TRUE)
403         exit(0);
404     else
405         exit(1);
406 }

407     if (OK == TRUE)
408         exit(0);
409     else
410         exit(1);
411 }

412     if (OK == TRUE)
413         exit(0);
414     else
415         exit(1);
416 }

417     if (OK == TRUE)
418         exit(0);
419     else
420         exit(1);
421 }

422     if (OK == TRUE)
423         exit(0);
424     else
425         exit(1);
426 }

427     if (OK == TRUE)
428         exit(0);
429     else
430         exit(1);
431 }

432     if (OK == TRUE)
433         exit(0);
434     else
435         exit(1);
436 }

437     if (OK == TRUE)
438         exit(0);
439     else
440         exit(1);
441 }

442     if (OK == TRUE)
443         exit(0);
444     else
445         exit(1);
446 }

447     if (OK == TRUE)
448         exit(0);
449     else
450         exit(1);
451 }

452     if (OK == TRUE)
453         exit(0);
454     else
455         exit(1);
456 }

457     if (OK == TRUE)
458         exit(0);
459     else
460         exit(1);
461 }

462     if (OK == TRUE)
463         exit(0);
464     else
465         exit(1);
466 }

467     if (OK == TRUE)
468         exit(0);
469     else
470         exit(1);
471 }

472     if (OK == TRUE)
473         exit(0);
474     else
475         exit(1);
476 }

477     if (OK == TRUE)
478         exit(0);
479     else
480         exit(1);
481 }

482     if (OK == TRUE)
483         exit(0);
484     else
485         exit(1);
486 }

487     if (OK == TRUE)
488         exit(0);
489     else
490         exit(1);
491 }

492     if (OK == TRUE)
493         exit(0);
494     else
495         exit(1);
496 }

497     if (OK == TRUE)
498         exit(0);
499     else
500         exit(1);
501 }

502     if (OK == TRUE)
503         exit(0);
504     else
505         exit(1);
506 }

507     if (OK == TRUE)
508         exit(0);
509     else
510         exit(1);
511 }

512     if (OK == TRUE)
513         exit(0);
514     else
515         exit(1);
516 }

517     if (OK == TRUE)
518         exit(0);
519     else
520         exit(1);
521 }

522     if (OK == TRUE)
523         exit(0);
524     else
525         exit(1);
526 }

527     if (OK == TRUE)
528         exit(0);
529     else
530         exit(1);
531 }

532     if (OK == TRUE)
533         exit(0);
534     else
535         exit(1);
536 }

537     if (OK == TRUE)
538         exit(0);
539     else
540         exit(1);
541 }

542     if (OK == TRUE)
543         exit(0);
544     else
545         exit(1);
546 }

547     if (OK == TRUE)
548         exit(0);
549     else
550         exit(1);
551 }

552     if (OK == TRUE)
553         exit(0);
554     else
555         exit(1);
556 }

557     if (OK == TRUE)
558         exit(0);
559     else
560         exit(1);
561 }

562     if (OK == TRUE)
563         exit(0);
564     else
565         exit(1);
566 }

567     if (OK == TRUE)
568         exit(0);
569     else
570         exit(1);
571 }

572     if (OK == TRUE)
573         exit(0);
574     else
575         exit(1);
576 }

577     if (OK == TRUE)
578         exit(0);
579     else
580         exit(1);
581 }

582     if (OK == TRUE)
583         exit(0);
584     else
585         exit(1);
586 }

587     if (OK == TRUE)
588         exit(0);
589     else
590         exit(1);
591 }

592     if (OK == TRUE)
593         exit(0);
594     else
595         exit(1);
596 }

597     if (OK == TRUE)
598         exit(0);
599     else
600         exit(1);
601 }

602     if (OK == TRUE)
603         exit(0);
604     else
605         exit(1);
606 }

607     if (OK == TRUE)
608         exit(0);
609     else
610         exit(1);
611 }

612     if (OK == TRUE)
613         exit(0);
614     else
615         exit(1);
616 }

617     if (OK == TRUE)
618         exit(0);
619     else
620         exit(1);
621 }

622     if (OK == TRUE)
623         exit(0);
624     else
625         exit(1);
626 }

627     if (OK == TRUE)
628         exit(0);
629     else
630         exit(1);
631 }

632     if (OK == TRUE)
633         exit(0);
634     else
635         exit(1);
636 }

637     if (OK == TRUE)
638         exit(0);
639     else
640         exit(1);
641 }

642     if (OK == TRUE)
643         exit(0);
644     else
645         exit(1);
646 }

647     if (OK == TRUE)
648         exit(0);
649     else
650         exit(1);
651 }

652     if (OK == TRUE)
653         exit(0);
654     else
655         exit(1);
656 }

657     if (OK == TRUE)
658         exit(0);
659     else
660         exit(1);
661 }

662     if (OK == TRUE)
663         exit(0);
664     else
665         exit(1);
666 }

667     if (OK == TRUE)
668         exit(0);
669     else
670         exit(1);
671 }

672     if (OK == TRUE)
673         exit(0);
674     else
675         exit(1);
676 }

677     if (OK == TRUE)
678         exit(0);
679     else
680         exit(1);
681 }

682     if (OK == TRUE)
683         exit(0);
684     else
685         exit(1);
686 }

687     if (OK == TRUE)
688         exit(0);
689     else
690         exit(1);
691 }

692     if (OK == TRUE)
693         exit(0);
694     else
695         exit(1);
696 }

697     if (OK == TRUE)
698         exit(0);
699     else
700         exit(1);
701 }

702     if (OK == TRUE)
703         exit(0);
704     else
705         exit(1);
706 }

707     if (OK == TRUE)
708         exit(0);
709     else
710         exit(1);
711 }

712     if (OK == TRUE)
713         exit(0);
714     else
715         exit(1);
716 }

717     if (OK == TRUE)
718         exit(0);
719     else
720         exit(1);
721 }

722     if (OK == TRUE)
723         exit(0);
724     else
725         exit(1);
726 }

727     if (OK == TRUE)
728         exit(0);
729     else
730         exit(1);
731 }

732     if (OK == TRUE)
733         exit(0);
734     else
735         exit(1);
736 }

737     if (OK == TRUE)
738         exit(0);
739     else
740         exit(1);
741 }

742     if (OK == TRUE)
743         exit(0);
744     else
745         exit(1);
746 }

747     if (OK == TRUE)
748         exit(0);
749     else
750         exit(1);
751 }

752     if (OK == TRUE)
753         exit(0);
754     else
755         exit(1);
756 }

757     if (OK == TRUE)
758         exit(0);
759     else
760         exit(1);
761 }

762     if (OK == TRUE)
763         exit(0);
764     else
765         exit(1);
766 }

767     if (OK == TRUE)
768         exit(0);
769     else
770         exit(1);
771 }

772     if (OK == TRUE)
773         exit(0);
774     else
775         exit(1);
776 }

777     if (OK == TRUE)
778         exit(0);
779     else
780         exit(1);
781 }

782     if (OK == TRUE)
783         exit(0);
784     else
785         exit(1);
786 }

787     if (OK == TRUE)
788         exit(0);
789     else
790         exit(1);
791 }

792     if (OK == TRUE)
793         exit(0);
794     else
795         exit(1);
796 }

797     if (OK == TRUE)
798         exit(0);
799     else
800         exit(1);
801 }

802     if (OK == TRUE)
803         exit(0);
804     else
805         exit(1);
806 }

807     if (OK == TRUE)
808         exit(0);
809     else
810         exit(1);
811 }

812     if (OK == TRUE)
813         exit(0);
814     else
815         exit(1);
816 }

817     if (OK == TRUE)
818         exit(0);
819     else
820         exit(1);
821 }

822     if (OK == TRUE)
823         exit(0);
824     else
825         exit(1);
826 }

827     if (OK == TRUE)
828         exit(0);
829     else
830         exit(1);
831 }

832     if (OK == TRUE)
833         exit(0);
834     else
835         exit(1);
836 }

837     if (OK == TRUE)
838         exit(0);
839     else
840         exit(1);
841 }

842     if (OK == TRUE)
843         exit(0);
844     else
845         exit(1);
846 }

847     if (OK == TRUE)
848         exit(0);
849     else
850         exit(1);
851 }

852     if (OK == TRUE)
853         exit(0);
854     else
855         exit(1);
856 }

857     if (OK == TRUE)
858         exit(0);
859     else
860         exit(1);
861 }

862     if (OK == TRUE)
863         exit(0);
864     else
865         exit(1);
866 }

867     if (OK == TRUE)
868         exit(0);
869     else
870         exit(1);
871 }

872     if (OK == TRUE)
873         exit(0);
874     else
875         exit(1);
876 }

877     if (OK == TRUE)
878         exit(0);
879     else
880         exit(1);
881 }

882     if (OK == TRUE)
883         exit(0);
884     else
885         exit(1);
886 }

887     if (OK == TRUE)
888         exit(0);
889     else
890         exit(1);
891 }

892     if (OK == TRUE)
893         exit(0);
894     else
895         exit(1);
896 }

897     if (OK == TRUE)
898         exit(0);
899     else
900         exit(1);
901 }

902     if (OK == TRUE)
903         exit(0);
904     else
905         exit(1);
906 }

907     if (OK == TRUE)
908         exit(0);
909     else
910         exit(1);
911 }

912     if (OK == TRUE)
913         exit(0);
914     else
915         exit(1);
916 }

917     if (OK == TRUE)
918         exit(0);
919     else
920         exit(1);
921 }

922     if (OK == TRUE)
923         exit(0);
924     else
925         exit(1);
926 }

927     if (OK == TRUE)
928         exit(0);
929     else
930         exit(1);
931 }

932     if (OK == TRUE)
933         exit(0);
934     else
935         exit(1);
936 }

937     if (OK == TRUE)
938         exit(0);
939     else
940         exit(1);
941 }

942     if (OK == TRUE)
943         exit(0);
944     else
945         exit(1);
946 }

947     if (OK == TRUE)
948         exit(0);
949     else
950         exit(1);
951 }

952     if (OK == TRUE)
953         exit(0);
954     else
955         exit(1);
956 }

957     if (OK == TRUE)
958         exit(0);
959     else
960         exit(1);
961 }

962     if (OK == TRUE)
963         exit(0);
964     else
965         exit(1);
966 }

967     if (OK == TRUE)
968         exit(0);
969     else
970         exit(1);
971 }

972     if (OK == TRUE)
973         exit(0);
974     else
975         exit(1);
976 }

977     if (OK == TRUE)
978         exit(0);
979     else
980         exit(1);
981 }

982     if (OK == TRUE)
983         exit(0);
984     else
985         exit(1);
986 }

987     if (OK == TRUE)
988         exit(0);
989     else
990         exit(1);
991 }

992     if (OK == TRUE)
993         exit(0);
994     else
995         exit(1);
996 }

997     if (OK == TRUE)
998         exit(0);
999     else
1000        exit(1);
1001 }
```

```

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

398 /* pick up args from standard input */
399
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;
449     N_args++;

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

```

```

455     }
456
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;
467         exit(exitstat);
468     }

469     /* insert arg if requested */
470
471     if (!ERR && INSERT) {
472
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 xc4, 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);
518             exitstat /= lcall(cmdname, arglist);
519         }
520     }

```

```

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 compliance,
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                 } /* FALLTHRU */
582             case '\0':

```

```

585         case WEOF:    /* Note WEOF == EOF */
586             if (escape) {
587                 EMSG(BADESCAPE);
588                 ERR = TRUE;
589                 return (NULL);
590             }
591             if (inquote) {
592                 EMSG(MISSQUOTE);
593                 ERR = TRUE;
594                 return (NULL);
595             }
596             N_lines++;
597             break;
598
599
600         case '"':
601             if (ZERO || escape || (inquote == 1)) {
602                 /* treat it literally */
603                 escape = 0;
604                 store_str(&arg, mbc, len);
605             }
606             else if (inquote == 2) {
607                 /* terminating double quote */
608                 inquote = 0;
609             }
610             else {
611                 /* starting quoted string */
612                 inquote = 2;
613             }
614             continue;
615
616         case '\'':
617             if (ZERO || escape || (inquote == 2)) {
618                 /* treat it literally */
619                 escape = 0;
620                 store_str(&arg, mbc, len);
621             }
622             else if (inquote == 1) {
623                 /* terminating single quote */
624                 inquote = 0;
625             }
626             else {
627                 /* starting quoted string */
628                 inquote = 1;
629             }
630             continue;
631
632
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             }
642             else {
643                 escape = 1;
644             }
645             continue;
646
647         default:
648             /*
649              * most times we will just want to store it */
650             if (inquote || escape || ZERO || !iswctype(c, blank)) {
651                 escape = 0;
652                 store_str(&arg, mbc, len);
653             }

```

```

651         continue;
652     } /* unquoted blank */
653     break;
654 }
655 */
656 /* At this point we are processing a complete argument.
657 */
658 if (strcmp(xarg, LEOF) == 0 && *LEOF != '\0') {
659     MORE = FALSE;
660     return (NULL);
661 }
662 if (c == WEOF) {
663     MORE = FALSE;
664 }
665 if (xarg[0] == '\0')
666     continue;
667 break;
668 }
669 */
670 return (xarg[0] == '\0' ? NULL : xarg);
671 }

672 /*
673 * errmsg(): print out an error message, and indicate failure globally.
674 */
675 Assumes that message has already been gettext()'d. It would be
676 nice if we could just do the gettext() here, but we can't, since
677 since xgettext(1M) wouldn't be able to pick up our error message.
678 */
679
680 /* PRINTFLIKE1 */
681 static void
682 errmsg(char *messages, ...)
683 {
684     va_list ap;
685
686     va_start(ap, messages);
687
688     (void) fprintf(stderr, "xargs: ");
689     (void) vfprintf(stderr, messages, ap);
690
691     va_end(ap);
692     OK = FALSE;
693 }
694 */

695 static int
696 echoargs()
697 {
698     char    **anarg;
699     char    **tanarg; /* tmp ptr */
700     int      i;
701     char    reply[LINE_MAX];
702
703     tanarg = anarg = arglist-1;
704
705     /*
706      * write out each argument, separated by a space. the tanarg
707      * nonsense is for xc4 testsuite compliance - so that an
708      * extra space isn't echoed after the last argument.
709      */
710     while (*++anarg) { /* while there's an argument */
711         ++tanarg; /* follow anarg */
712         (void) write(2, *anarg, strlen(*anarg));
713
714         if (*++tanarg) { /* if there's another argument: */
715             /* unquoted blank */
716             break;
717         }
718     }
719 }
720
721 if (PROMPT == -1) {
722     (void) write(2, "\n", 1);
723     return (TRUE);
724 }
725
726 (void) write(2, "...", 4); /* ask the user for input */
727
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;
736
737 /* flush remainder of line if necessary */
738 if (i == LINE_MAX) {
739     char    bitbucket;
740
741     while ((read(PROMPT, &bitbucket, 1) > 0) && (bitbucket != '\n'))
742         ;
743 }
744
745 return (yes_check(reply));
746 }

747 static char *
748 insert(char *pattern, char *subst)
749 {
750     static char    buffer[MAXSBUF+1];
751     int      len, ipatlen;
752     char    *pat;
753     char    *bufend;
754     char    *pbuf;
755
756     len = strlen(subst);
757     ipatlen = strlen(INSPAT) - 1;
758     pat = pattern - 1;
759     pbuf = buffer;
760     bufend = &buffer[MAXSBUF];
761
762     while (++pat) {
763         if (strncmp(pat, INSPAT, ipatlen) == 0) {
764             if (pbuf + len >= bufend) {
765                 break;
766             } else {
767                 (void) strcpy(pbuf, subst);
768                 pat += ipatlen;
769                 pbuf += len;
770             }
771         } else {
772             *pbuf++ = *pat;
773             if (pbuf >= bufend)
774                 break;
775         }
776     }
777
778     if (!*pat) {
779         *pbuf = '\0';
780         return (buffer);
781     }
782 }
```

```

783     } else {
784         errmsg(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;
797
798     skel = p->p_skel;
799     sub = *ARGV;
800     newarg = insert(skel, sub);
801     if (ERR)
802         return;
803
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;
828
829     i = 0;
830     while (i < MB_CUR_MAX) {
831
832         if ((c = fgetc(stdin)) == EOF) {
833
834             if (i == 0) {
835                 /* TRUE EOF has been reached */
836                 return (WEOF);
837             }
838
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;
846
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
874 lcall(char *sub, char **subargs)
875 {
876     int      retry = 0;
877     pid_t   child;
878     int retcode, retry = 0;
879     pid_t iwait, child;
880
881     for (;;) {
882         switch (child = fork()) {
883         default:
884             (void) procs_store(child);
885             (void) procs_wait(0);
886             return;
887             while ((iwait = wait(&retcode)) != child &&
888                   iwait != (pid_t)-1)
889             if (iwait == (pid_t)-1) {
890                 PERR(WAITFAIL);
891                 exit(122);
892                 /* NOTREACHED */
893             }
894             if (WIFSIGNALED(retcode)) {
895                 EMSG2(CHILDSIG, WTERMSIG(retcode));
896                 exit(125);
897                 /* NOTREACHED */
898             }
899             if ((WEXITSTATUS(retcode) & 0377) == 0377) {
900                 EMSG(CHILDFAIL);
901                 exit(124);
902                 /* NOTREACHED */
903             }
904             return (WEXITSTATUS(retcode));
905     case 0:
906         (void) execvp(sub, subargs);
907         PERR(EXECFAIL);
908         if (errno == EACCES)
909             exit(126);
910         exit(127);
911         /* NOTREACHED */
912     case -1:

```

```

893         if (errno != EAGAIN && retry++ < FORK_RETRY) {
894             PERR(FORKFAIL);
895             exit(123);
896         }
897         (void) sleep(1);
898     }
899 }
900 }

902 static void
903 procs_malloc(void)
904 {
905     int i;
906
907     procs = (pid_t *)malloc(MAXPROCS * sizeof(pid_t));
908     if (procs == NULL) {
909         PERR(MALLOCFAIL);
910         exit(1);
911     }
912
913     for (i = 0; i < MAXPROCS; i++) {
914         procs[i] = (pid_t)(0);
915     }
916 }

918 static int
919 procs_find(pid_t child)
920 {
921     int i;
922
923     for (i = 0; i < MAXPROCS; i++) {
924         if (procs[i] == child) {
925             return (i);
926         }
927     }
928
929     return (-1);
930 }

932 static void
933 procs_store(pid_t child)
934 {
935     int i;
936
937     i = procs_find((pid_t)(0));
938     if (i < 0) {
939         PERR(NOCHILDSLOT);
940         exit(1);
941     }
942     procs[i] = child;
943     n_procs++;
944 }

946 static int
947 procs_delete(pid_t child)
948 {
949     int i;
950
951     i = procs_find(child);
952     if (i < 0) {
953         return (0);
954     }
955     procs[i] = (pid_t)(0);
956     n_procs--;
957     return (1);
958 }

```

```

960 static pid_t
961 procs_waitpid(int blocking, int *stat_loc)
962 {
963     pid_t child;
964     int options;
965
966     if (n_procs == 0) {
967         errno = ECHILD;
968         return (-1);
969     }
970
971     options = (blocking) ? 0 : WNOHANG;
972
973     while ((child = waitpid(-1, stat_loc, options)) > 0) {
974         if (procs_delete(child)) {
975             break;
976         }
977     }
978
979     return (child);
980 }

982 static void
983 procs_wait(int blocking)
984 {
985     pid_t child;
986     int stat_loc;
987
988     while ((child = procs_waitpid(blocking || (n_procs >= MAXPROCS) ? 1 : 0,
989                                     if (WIFSIGNALED(stat_loc)) {
990                                         EMSG2(CHILDSIG, WTERMSIG(stat_loc));
991                                         exit(125);
992                                         /* NOTREACHED */
993                                     } else if ((WEXITSTATUS(stat_loc) & 0377) == 0377) {
994                                         EMSG(CHILDFAIL);
995                                         exit(124);
996                                         /* NOTREACHED */
997                                     } else {
998                                         exitstat |= WEXITSTATUS(stat_loc);
999                                     }
1000
1001         if (child == (pid_t)(-1) && errno != ECHILD) {
1002             EMSG(WAITFAIL);
1003             exit(122);
1004             /* NOTREACHED */
1005         }
1006     }
1007 }
1008 #endif /* ! codereview */

1010 static void
1011 usage()
1012 {
1013     ermsg_(USAGEMSG);
1014     OK = FALSE;
1015 }

1019 /*
1020  * parseargs():      modify the args
1021  *                   since the -e, -i and -l flags all take optional subarguments,
1022  *                   and getopt(3C) is clueless about this nonsense, we change the
1023  *                   our local argument count and strings to separate this out,
1024  *                   and make it easier to handle via getopt(3c).

```

```

1025 *
1026 *      -e      -> "-e \""
1027 *      -e3     -> "-e \"3\""
1028 *      -Estr   -> "-E \"str\""
1029 *      -i      -> "-i \"{}\""
1030 *      -irep   -> "-i \"rep\""
1031 *      -l      -> "-i \"1\""
1032 *      -l10    -> "-i \"10\""
1033 *
1034 *      since the -e, -i and -l flags all take optional subarguments,
1035 */
1036 static void
1037 parseargs(int ac, char **av)
1038 {
1039     int i;                      /* current argument          */
1040     int cflag;                  /* 0 = not processing cmd arg */
1041
1042     if ((mav = malloc((ac * 2 + 1) * sizeof (char *))) == NULL) {
1043         PERR(MALLOCFAIL);
1044         exit(1);
1045     }
1046
1047     /* for each argument, see if we need to change things:          */
1048     for (i = mac = cflag = 0; (av[i] != NULL) && i < ac; i++, mac++) {
1049         if ((mav[mac] = strdup(av[i])) == NULL) {
1050             PERR(MALLOCFAIL);
1051             exit(1);
1052         }
1053
1054         /* -- has been found or argument list is fully processes */
1055         if (cflag)
1056             continue;
1057
1058         /*
1059         * if we're doing special processing, and we've got a flag
1060         */
1061         else if ((av[i][0] == '-') && (av[i][1] != NULL)) {
1062             char *def;
1063
1064             switch (av[i][1]) {
1065                 case 'e':
1066                     def = ""; /* -e with no arg turns off eof */
1067                     goto process_special;
1068
1069                 case 'i':
1070                     def = INSPAT_STR;
1071                     goto process_special;
1072
1073         process_special:
1074             /*
1075             * if there's no sub-option, we *must* add
1076             * a default one. this is because xargs must
1077             * be able to distinguish between a valid
1078             * suboption, and a command name.
1079             */
1080             if (av[i][2] == NULL) {
1081                 mav[++mac] = strdup(def);
1082             } else {
1083                 /* clear out our version: */
1084                 mav[mac][2] = NULL;
1085                 mav[++mac] = strdup(&av[i][2]);
1086             }
1087             if (mav[mac] == NULL) {
1088                 PERR(MALLOCFAIL);
1089                 exit(1);
1090         }

```

```

1091                                         break;
1092
1093     /* flags with required subarguments: */
1094
1095     /*
1096     * there are two separate cases here. either the
1097     * flag can have the normal XCU4 handling
1098     * (of the form: -X subargument); or it can have
1099     * the old solaris 2.[0-4] handling (of the
1100     * form: -Xsubargument). in order to maintain
1101     * backwards compatibility, we must support the
1102     * latter case. we handle the latter possibility
1103     * first so both the old solaris way of handling
1104     * and the new XCU4 way of handling things are allowed.
1105
1106     case 'n': /* FALLTHROUGH */
1107     case 'P': /* FALLTHROUGH */
1108 #endif /* ! codereview */
1109     case 's': /* FALLTHROUGH */
1110     case 'E': /* FALLTHROUGH */
1111     case 'I': /* FALLTHROUGH */
1112     case 'L':
1113         /*
1114         * if the second character isn't null, then
1115         * the user has specified the old syntax.
1116         * we move the subargument into our
1117         * mod'd argument list.
1118
1119         if (av[i][2] != NULL) {
1120             /* first clean things up: */
1121             mav[mac][2] = NULL;
1122
1123             /* now add the separation: */
1124             ++mac; /* inc to next mod'd arg */
1125             if ((mav[mac] = strdup(&av[i][2])) ==
1126                 NULL) {
1127                 PERR(MALLOCFAIL);
1128                 exit(1);
1129             }
1130             break;
1131         }
1132         i++;
1133         mac++;
1134
1135         if (av[i] == NULL) {
1136             mav[mac] = NULL;
1137             return;
1138         }
1139         if ((mav[mac] = strdup(av[i])) == NULL) {
1140             PERR(MALLOCFAIL);
1141             exit(1);
1142         }
1143         break;
1144
1145     /* flags */
1146     case 'p':
1147     case 't':
1148     case 'x':
1149     case 'o':
1150         break;
1151
1152     case '-' :
1153     default:
1154         /*
1155         * here we've hit the cmd argument. so
1156         * 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 }
1171
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\'\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}\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\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\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
78 EOF 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\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\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\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-i\fR, and \fB-n\fR
121 options are mutually exclusive; the last one specified takes effect.
122 .RE

124 .sp
125 .ne 2
```

```

126 .na
127 \fB\fB-1[\fR\fInumber\fR\fB]\fR\fR
128 .ad
129 .RS 15n
130 (The letter ell.) This option is equivalent to \fB-L\fR \fInumber\fR. If
131 \fInumber\fR is omitted, \fB1\fR is assumed. Option \fB-x\fR is forced on.
132 .RE

134 .sp
135 .ne 2
136 .na
137 \fB\fB-n\fR \fInumber\fR\fR
138 .ad
139 .RS 15n
140 Invokes \fIutility\fR using as many standard input arguments as possible, up to
141 \fInumber\fR (a positive decimal integer) arguments maximum. Fewer arguments
142 are used if:
143 .RS +4
144 .TP
145 .ie t \(\bu
146 .el o
147 The command line length accumulated exceeds the size specified by the \fB-s\fR
148 option (or \fB{LINE_MAX}\fR if there is no \fB-s\fR option), or
149 .RE
150 .RS +4
151 .TP
152 .ie t \(\bu
153 .el o
154 The last iteration has fewer than \fInumber\fR, but not zero, operands
155 remaining.
156 .RE
157 .RE

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

172 .sp
173 .ne 2
174 .na
175 \fB\fB-P\fR \fImaxprocs\fR\fR
176 .ad
177 .RS 15n
178 Invokes \fIutility\fR using at most \fImaxprocs\fR (a positive decimal integer)
179 parallel child processes.
180 #endif /* ! codereview */
181 .RE

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

```

```

192 .RS +4
193 .TP
194 .ie t \(\bu
195 .el o
196 The total number of arguments exceeds that specified by the \fB-n\fR option, or
197 .RE
198 .RS +4
199 .TP
200 .ie t \(\bu
201 .el o
202 The total number of lines exceeds that specified by the \fB-L\fR option, or
203 .RE
204 .RS +4
205 .TP
206 .ie t \(\bu
207 .el o
208 End of file is encountered on standard input before \fIsize\fR bytes are
209 accumulated.
210 .RE
211 Values of \fIsize\fR up to at least \fB{LINE_MAX}\fR bytes are supported,
212 provided that the constraints specified in DESCRIPTION are met. It is not
213 considered an error if a value larger than that supported by the implementation
214 or exceeding the constraints specified in DESCRIPTION is specified. \fBxargs\fR
215 uses the largest value it supports within the constraints.
216 .RE

218 .sp
219 .ne 2
220 .na
221 \fB\fB-t\fR\fR\fR
222 .ad
223 .RS 6n
224 Enables trace mode. Each generated command line is written to standard error
225 just prior to invocation.
226 .RE

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

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

250 .SH OPERANDS
251 .sp
252 .LP
253 The following operands are supported:
254 .sp
255 .ne 2
256 .na
257 \fB\fIutility\fR\fR\fR

```

```

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

267 .sp
268 .ne 2
269 .na
270 \fB\fIargument\fR\fR
271 .ad
272 .RS 12n
273 An initial option or operand for the invocation of \fIutility\fR.
274 .RE

276 .SH USAGE
277 .sp
278 .LP
279 The \fB255\fR exit status allows a utility being used by \fBxargs\fR to tell
280 \fBxargs\fR to terminate if it knows no further invocations using the current
281 data stream succeeds. Thus, \fIutility\fR should explicitly \fBexit\fR with an
282 appropriate value to avoid accidentally returning with \fB255\fR.
283 .sp
284 .LP
285 Notice that input is parsed as lines. Blank characters separate arguments. If
286 \fBxargs\fR is used to bundle output of commands like \fBfind\fR \fBfind\fR
287 \fBprint\fR or \fBls\fR into commands to be executed, unexpected results are
288 likely if any filenames contain any blank characters or newline characters.
289 This can be fixed by using \fBfind\fR to call a script that converts each file
290 found into a quoted string that is then piped to \fBxargs\fR. Notice that the
291 quoting rules used by \fBxargs\fR are not the same as in the shell. They were
292 not made consistent here because existing applications depend on the current
293 rules and the shell syntax is not fully compatible with it. An easy rule that
294 can be used to transform any string into a quoted form that \fBxargs\fR
295 interprets correctly is to precede each character in the string with a
296 backslash (\fBe\fR).
297 .sp
298 .LP
299 On implementations with a large value for \fB{ARG_MAX}\fR, \fBxargs\fR can
300 produce command lines longer than \fB{LINE_MAX}\fR. For invocation of
301 utilities, this is not a problem. If \fBxargs\fR is being used to create a text
302 file, users should explicitly set the maximum command line length with the
303 \fB-s\fR option.
304 .sp
305 .LP
306 The \fBxargs\fR utility returns exit status \fB127\fR if an error occurs so
307 that applications can distinguish "failure to find a utility" from "invoked
308 utility exited with an error indication." The value \fB127\fR was chosen
309 because it is not commonly used for other meanings; most utilities use small
310 values for "normal error conditions" and the values above \fB128\fR can be
311 confused with termination due to receipt of a signal. The value \fB126\fR was
312 chosen in a similar manner to indicate that the utility could be found, but not
313 invoked.
314 .SH EXAMPLES
315 .LP
316 \fBExample 1\fR Using the xargs command
317 .sp
318 .LP
319 The following example moves all files from directory \fB$1\fR to directory
320 \fB$2\fR, and echo each move command just before doing it:
321 .sp
322 .in +2

```

```

324 .nf
325 example% \fBls $1 | xargs -I {} -t mv ${1/{} ${2/{} }\fR
326 .fi
327 .in -2
328 .sp

330 .sp
331 .LP
332 The following command combines the output of the parenthesised commands onto
333 one line, which is then written to the end of file \fBlog\fR:
335 .sp
336 .in +2
337 .nf
338 example% \fB(logname; date; printf "%s\n" "$0 $*") | xargs >>log\fR
339 .fi
340 .in -2
341 .sp

343 .sp
344 .LP
345 The following command invokes \fBdiff\fR with successive pairs of arguments
346 originally typed as command line arguments (assuming there are no embedded
347 blank characters in the elements of the original argument list):
349 .sp
350 .in +2
351 .nf
352 example% \fBprintf "%s\n" "$*" | xargs -n 2 -x diff\fR
353 .fi
354 .in -2
355 .sp

357 .sp
358 .LP
359 The user is asked which files in the current directory are to be archived. The
360 files are archived into \fBarch\fR \fB; \fR a, one at a time, or b, many at a
361 time:
363 .sp
364 .in +2
365 .nf
366 example% \fBls | xargs -p -L 1 ar -r arch
367 ls | xargs -p -L 1 | xargs ar -r arch\fR
368 .fi
369 .in -2
370 .sp

372 .sp
373 .LP
374 The following executes with successive pairs of arguments originally typed as
375 command line arguments:
377 .sp
378 .in +2
379 .nf
380 example% \fBecho $* | xargs -n 2 diff\fR
381 .fi
382 .in -2
383 .sp

385 .SH ENVIRONMENT VARIABLES
386 .sp
387 .LP
388 See \fB environ\fR(5) for descriptions of the following environment variables
389 that affect the execution of \fBxargs\fR: \fBLANG\fR, \fBLC_ALL\fR,

```

```

390 \fBLC_COLLATE\fR, \fBLC_CTYPE\fR, \fBLC_MESSAGES\fR, and \fBNLSPATH\fR.
391 .sp
392 .ne 2
393 .na
394 \fB\fBPATH\fR\fR
395 .ad
396 .RS 8n
397 Determine the location of \fIutility\fR.
398 .RE

400 .sp
401 .LP
402 Affirmative responses are processed using the extended regular expression
403 defined for the \fByesexpr\fR keyword in the \fBLC_MESSAGES\fR category of the
404 user's locale. The locale specified in the \fBLC_COLLATE\fR category defines
405 the behavior of ranges, equivalence classes, and multi-character collating
406 elements used in the expression defined for \fByesexpr\fR. The locale specified
407 in \fBLC_CTYPE\fR determines the locale for interpretation of sequences of
408 bytes of text data a characters, the behavior of character classes used in the
409 expression defined for the \fByesexpr\fR. See \fBlocale\fR(5).
410 .SH EXIT STATUS
411 .sp
412 .LP
413 The following exit values are returned:
414 .sp
415 .ne 2
416 .na
417 \fB\fB0\fR\fR
418 .ad
419 .RS 12n
420 All invocations of \fIutility\fR returned exit status \fB0\fR.
421 .RE

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

434 .sp
435 .ne 2
436 .na
437 \fB\fB126\fR\fR
438 .ad
439 .RS 12n
440 The utility specified by \fIutility\fR was found but could not be invoked.
441 .RE

443 .sp
444 .ne 2
445 .na
446 \fB\fB127\fR\fR
447 .ad
448 .RS 12n
449 The utility specified by \fIutility\fR could not be found.
450 .RE

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

```

```

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

465 .sp
466 .TS
467 box;
468 c | c
469 l | l .
470 ATTRIBUTE TYPE ATTRIBUTE VALUE
471 -
472 CSI Enabled
473 -
474 Interface Stability Standard
475 .TE

477 .SH SEE ALSO
478 .sp
479 .LP
480 \fBecho\fR(1), \fBshell_builtin\fR(1), \fBattributes\fR(5), \fBenvirons\fR(5),
481 \fBstandards\fR(5)

```