

```

*****
90291 Wed May 20 12:04:50 2015
new/usr/src/cmd/make/bin/main.cc
make: remove more distributed mode code
*****
_____unchanged_portion_omitted_____

136 extern Name          normalize_name(register wchar_t *name_string, register i

138 extern int           main(int, char * []);

140 static void          append_makeflags_string(Name, String);
141 static void          doalarm(int);
142 static void          enter_argv_values(int , char **, ASCII_Dyn_Array *);
143 static void          make_targets(int, char **, Boolean);
144 static int           parse_command_option(char);
145 static void          read_command_options(int, char **);
146 static void          read_environment(Boolean);
147 static void          read_files_and_state(int, char **);
148 static Boolean       read_makefile(Name, Boolean, Boolean, Boolean);
149 static void          report_recursion(Name);
150 static void          set_sgs_support(void);
151 static void          setup_for_projectdir(void);
152 static void          setup_makeflags_argv(void);
153 static void          report_dir_enter_leave(Boolean entering);

155 extern void expand_value(Name, register String , Boolean);

157 static const char    verstring[] = "illumos make";

159 jmp_buf jmpbuffer;
160 extern nl_catd catd;

162 /*
163 *      main(argc, argv)
164 *
165 *      Parameters:
166 *          argc          You know what this is
167 *          argv          You know what this is
168 *
169 *      Static variables used:
170 *          list_all_targets      make -T seen
171 *          trace_status         make -p seen
172 *
173 *      Global variables used:
174 *          debug_level          Should we trace make actions?
175 *          keep_state          Set if .KEEP_STATE seen
176 *          makeflags           The Name "MAKEFLAGS", used to get macro
177 *          remote_command_name Name of remote invocation cmd ("on")
178 *          running_list        List of parallel running processes
179 *          stdout_stderr_same  true if stdout and stderr are the same
180 *          auto_dependencies   The Name "SUNPRO_DEPENDENCIES"
181 *          temp_file_directory Set to the dir where we create tmp file
182 *          trace_reader        Set to reflect tracing status
183 *          working_on_targets  Set when building user targets
184 */
185 int
186 main(int argc, char *argv[])
187 {
188     /*
189     * cp is a -> to the value of the MAKEFLAGS env var,
190     * which has to be regular chars.
191     */
192     register char    *cp;
193     char             make_state_dir[MAXPATHLEN];
194     Boolean          parallel_flag = false;

```

```

195     char             *prognameptr;
196     char             *slash_ptr;
197     mode_t           um;
198     int              i;
199     struct itimerval value;
200     char             def_dmakerc_path[MAXPATHLEN];
201     Name             dmake_name, dmake_name2;
202     Name             dmake_value, dmake_value2;
203     Property         prop, prop2;
204     struct stat      statbuf;
205     int              statval;

207     struct stat      out_stat, err_stat;
208     hostid = gethostid();
209     bsd_signals();

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

214 #ifndef DMAKE_STATISTICS
215     if (getenv(NOCATGETS("DMAKE_STATISTICS"))) {
216         getname_stat = true;
217     }
218 #endif

220     catd = catopen(AVO_DOMAIN_DMAKE, NL_CAT_LOCALE);

222 // ---> fprintf(stderr, catgets(catd, 15, 666, "--- SUN make ---\n"));

225 /*
226 * I put libmksdmsil8n_init() under #ifdef because it requires avo_i18n_init()
227 * from avo_util library.
228 */
229     libmksdmsil8n_init();

232     textdomain(NOCATGETS("SUNW_SPRO_MAKE"));

234     g_argc = argc;
235     g_argv = (char **) malloc((g_argc + 1) * sizeof(char *));
236     for (i = 0; i < argc; i++) {
237         g_argv[i] = argv[i];
238     }
239     g_argv[i] = NULL;

241     /*
242     * Set argv_zero_string to some form of argv[0] for
243     * recursive MAKE builds.
244     */

246     if (*argv[0] == (int) slash_char) {
247         /* argv[0] starts with a slash */
248         argv_zero_string = strdup(argv[0]);
249     } else if (strchr(argv[0], (int) slash_char) == NULL) {
250         /* argv[0] contains no slashes */
251         argv_zero_string = strdup(argv[0]);
252     } else {
253         /*
254         * argv[0] contains at least one slash,
255         * but doesn't start with a slash
256         */
257         char    *tmp_current_path;
258         char    *tmp_string;

260         tmp_current_path = get_current_path();

```

```

261     tmp_string = getmem(strlen(tmp_current_path) + 1 +
262                       strlen(argv[0]) + 1);
263     (void) sprintf(tmp_string,
264                  "%s/%s",
265                  tmp_current_path,
266                  argv[0]);
267     argv_zero_string = strdup(tmp_string);
268     retmem_mb(tmp_string);
269 }

271 /*
272  * The following flags are reset if we don't have the
273  * (.nse_depinfo or .make.state) files locked and only set
274  * AFTER the file has been locked. This ensures that if the user
275  * interrupts the program while file_lock() is waiting to lock
276  * the file, the interrupt handler doesn't remove a lock
277  * that doesn't belong to us.
278  */
279 make_state_lockfile = NULL;
280 make_state_locked = false;

283 /*
284  * look for last slash char in the path to look at the binary
285  * name. This is to resolve the hard link and invoke make
286  * in svr4 mode.
287  */

289 /* Sun OS make standart */
290 svr4 = false;
291 posix = false;
292 if(!strcmp(argv_zero_string, NOCATGETS("/usr/xpg4/bin/make"))) {
293     svr4 = false;
294     posix = true;
295 } else {
296     prognameptr = strrchr(argv[0], '/');
297     if(prognameptr) {
298         prognameptr++;
299     } else {
300         prognameptr = argv[0];
301     }
302     if(!strcmp(prognameptr, NOCATGETS("svr4.make"))) {
303         svr4 = true;
304         posix = false;
305     }
306 }
307 if (getenv(USE_SVR4_MAKE) || getenv(NOCATGETS("USE_SVID"))){
308     svr4 = true;
309     posix = false;
310 }

312 /*
313  * Find the dmake_compat_mode: posix, sun, svr4, or gnu_style, .
314  */
315 char * dmake_compat_mode_var = getenv(NOCATGETS("SUN_MAKE_COMPAT_MODE"));
316 if (dmake_compat_mode_var != NULL) {
317     if (0 == strcasecmp(dmake_compat_mode_var, NOCATGETS("GNU"))) {
318         gnu_style = true;
319     }
320     //svr4 = false;
321     //posix = false;
322 }

324 /*
325  * Temporary directory set up.
326  */

```

```

327     char * tmpdir_var = getenv(NOCATGETS("TMPDIR"));
328     if (tmpdir_var != NULL && *tmpdir_var == '/' && strlen(tmpdir_var) < MAX
329         strcpy(mbs_buffer, tmpdir_var);
330         for (tmpdir_var = mbs_buffer+strlen(mbs_buffer);
331             *(--tmpdir_var) == '/' && tmpdir_var > mbs_buffer;
332             *tmpdir_var = '\0');
333     if (strlen(mbs_buffer) + 32 < MAXPATHLEN) { /* 32 = strlen("/dma
334         sprintf(mbs_buffer2, NOCATGETS("%s/dmake.tst.%d.XXXXXX")
335             mbs_buffer, getpid());
336         int fd = mkstemp(mbs_buffer2);
337         if (fd >= 0) {
338             close(fd);
339             unlink(mbs_buffer2);
340             tmpdir = strdup(mbs_buffer);
341         }
342     }
343 }

345 /* find out if stdout and stderr point to the same place */
346 if (fstat(1, &out_stat) < 0) {
347     fatal(catgets(catd, 1, 165, "fstat of standard out failed: %s"),
348         );
349     if (fstat(2, &err_stat) < 0) {
350         fatal(catgets(catd, 1, 166, "fstat of standard error failed: %s"
351             );
352     }
353     if ((out_stat.st_dev == err_stat.st_dev) &&
354         (out_stat.st_ino == err_stat.st_ino)) {
355         stdout_stderr_same = true;
356     } else {
357         stdout_stderr_same = false;
358     }
359 }
360 /* Make the vroot package scan the path using shell semantics */
361 set_path_style(0);

362 setup_char_semantics();

363 setup_for_projectdir();

365 /*
366  * If running with .KEEP_STATE, curdir will be set with
367  * the connected directory.
368  */
369 (void) atexit(cleanup_after_exit);

371 load_cached_names();

373 /*
374  *
375  */
376 Set command line flags
377 setup_makeflags_argv();
378 read_command_options(mf_argc, mf_argv);
379 read_command_options(argc, argv);
380 if (debug_level > 0) {
381     cp = getenv(makeflags->string_mb);
382     (void) printf(catgets(catd, 1, 167, "MAKEFLAGS value: %s\n"), cp);
383 }

384 setup_interrupt(handle_interrupt);

386 read_files_and_state(argc, argv);

388 /*
389  * Find the dmake_output_mode: TXT1, TXT2 or HTML1.
390  */
391 MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_OUTPUT_MODE"));
392 dmake_name2 = GETNAME(wcs_buffer, FIND_LENGTH);

```

```

393     prop2 = get_prop(dmake_name2->prop, macro_prop);
394     if (prop2 == NULL) {
395         /* DMAKE_OUTPUT_MODE not defined, default to TXT1 mode */
396         output_mode = txt1_mode;
397     } else {
398         dmake_value2 = prop2->body.macro.value;
399         if ((dmake_value2 == NULL) ||
400             (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("TXT1")))) {
401             output_mode = txt1_mode;
402         } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("TXT2")))
403             output_mode = txt2_mode;
404         } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("HTML1")))
405             output_mode = html1_mode;
406         } else {
407             warning(catgets(catd, 1, 352, "Unsupported value '%s' fo
408                 dmake_value2->string_mb);
409         }
410     }
411     /*
412     * Find the dmake_mode: parallel, or serial.
413     * Find the dmake_mode: distributed, parallel, or serial.
414     */
415     if ((!pmake_cap_r_specified) &&
416         (!pmake_machinesfile_specified)) {
417         MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_MODE"));
418         dmake_name2 = GETNAME(wcs_buffer, FIND_LENGTH);
419         prop2 = get_prop(dmake_name2->prop, macro_prop);
420         if (prop2 == NULL) {
421             /* DMAKE_MODE not defined, default to parallel mode */
422             dmake_mode_type = parallel_mode;
423             /* DMAKE_MODE not defined, default to distributed mode */
424             dmake_mode_type = distributed_mode;
425             no_parallel = false;
426         } else {
427             dmake_value2 = prop2->body.macro.value;
428             if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("parallel"))) {
429                 if ((dmake_value2 == NULL) ||
430                     (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("distributed"))
431                         dmake_mode_type = distributed_mode;
432                         no_parallel = false;
433                     } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("parallel
434                         dmake_mode_type = parallel_mode;
435                         no_parallel = false;
436                     } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("serial")
437                         dmake_mode_type = serial_mode;
438                         no_parallel = true;
439                     } else {
440                         fatal(catgets(catd, 1, 307, "Unknown dmake mode argument
441                     }
442                 }
443             }
444         }
445     }
446     if ((!list_all_targets) &&
447         (report_dependencies_level == 0)) {
448         /*
449         * Check to see if either DMAKE_RCFILE or DMAKE_MODE is defined.
450         * They could be defined in the env, in the makefile, or on the
451         * command line.
452         * If neither is defined, and $(HOME)/.dmakerc does not exist,
453         * then print a message, and default to parallel mode.
454         */
455         if (dmake_mode_type == distributed_mode) {
456             if (dmake_mode_type == parallel_mode;
457                 no_parallel = false;
458             }
459         }
460     }
461 }

```

```

438     parallel_flag = true;
439     putenv(strdup(NOCATGETS("DMAKE_CHILD=TRUE")));
440
441 //
442 // If dmake is running with -t option, set dmake_mode_type to serial.
443 // This is done because doname() calls touch_command() that runs serially.
444 // If we do not do that, maketool will have problems.
445 //
446     if (touch) {
447         dmake_mode_type = serial_mode;
448         no_parallel = true;
449     }
450
451     /*
452     * Check whether stdout and stderr are physically same.
453     * This is in order to decide whether we need to redirect
454     * stderr separately from stdout.
455     * This check is performed only if __DMAKE_SEPARATE_STDERR
456     * is not set. This variable may be used in order to preserve
457     * the 'old' behaviour.
458     */
459     out_err_same = true;
460     char * dmake_sep_var = getenv(NOCATGETS("__DMAKE_SEPARATE_STDERR"));
461     if (dmake_sep_var == NULL || (0 != strcmp(dmake_sep_var, NOCATGETS("
462         struct stat stdout_stat;
463         struct stat stderr_stat;
464         if (fstat(1, &stdout_stat) == 0)
465             && (fstat(2, &stderr_stat) == 0))
466         {
467             if ( (stdout_stat.st_dev != stderr_stat.st_dev)
468                 || (stdout_stat.st_ino != stderr_stat.st_ino) )
469                 out_err_same = false;
470         }
471     }
472 }
473
474
475 //
476 /*
477 * Enable interrupt handler for alarms
478 */
479 (void) bsd_signal(SIGALRM, (SIG_PF)doalarm);
480
481 //
482 * Check if make should report
483 */
484 if (getenv(sunpro_dependencies->string_mb) != NULL) {
485     FILE *report_file;
486
487     report_dependency("");
488     report_file = get_report_file();
489     if ((report_file != NULL) && (report_file != (FILE*)-1)) {
490         (void) fprintf(report_file, "\n");
491     }
492 }
493
494 //
495 * Make sure SUNPRO_DEPENDENCIES is exported (or not) properly.
496 */
497 if (keep_state) {
498     maybe_append_prop(sunpro_dependencies, macro_prop)->
499         body.macro.exported = true;
500 } else {
501     maybe_append_prop(sunpro_dependencies, macro_prop)->
502         body.macro.exported = false;

```

```

503     }
505     working_on_targets = true;
506     if (trace_status) {
507         dump_make_state();
508         fclose(stdout);
509         fclose(stderr);
510         exit_status = 0;
511         exit(0);
512     }
513     if (list_all_targets) {
514         dump_target_list();
515         fclose(stdout);
516         fclose(stderr);
517         exit_status = 0;
518         exit(0);
519     }
520     trace_reader = false;
522     /*
523     * Set temp_file_directory to the directory the .make.state
524     * file is written to.
525     */
526     if ((slash_ptr = strrchr(make_state->string_mb, (int) slash_char)) == NU
527         temp_file_directory = strdup(get_current_path());
528     } else {
529         *slash_ptr = (int) nul_char;
530         (void) strcpy(make_state_dir, make_state->string_mb);
531         *slash_ptr = (int) slash_char;
532         /* when there is only one slash and it's the first
533         ** character, make_state_dir should point to '/'.
534         */
535         if (make_state_dir[0] == '\0') {
536             make_state_dir[0] = '/';
537             make_state_dir[1] = '\0';
538         }
539         if (make_state_dir[0] == (int) slash_char) {
540             temp_file_directory = strdup(make_state_dir);
541         } else {
542             char    tmp_current_path2[MAXPATHLEN];
543             (void) sprintf(tmp_current_path2,
544                 "%s/%s",
545                 get_current_path(),
546                 make_state_dir);
547             temp_file_directory = strdup(tmp_current_path2);
548         }
549     }
550 }
553     report_dir_enter_leave(true);
555     make_targets(argc, argv, parallel_flag);
557     report_dir_enter_leave(false);
559     if (build_failed_ever_seen) {
560         if (posix) {
561             exit_status = 1;
562         }
563         exit(1);
564     }
565     exit_status = 0;
566     exit(0);
567     /* NOTREACHED */
568 }

```

unchanged portion omitted

```

727 /*
728 *     handle_interrupt()
729 *
730 *     This is where C-C traps are caught.
731 *
732 *     Parameters:
733 *
734 *     Global variables used (except DMake 1.0):
735 *         current_target     Sometimes the current target is removed
736 *         do_not_exec_rule   But not if -n is on
737 *         quest              or -q
738 *         running_list       List of parallel running processes
739 *         touch              Current target is not removed if -t on
740 */
741 void
742 handle_interrupt(int)
743 {
744     Property          member;
745     Running           rp;
747     (void) fflush(stdout);
748     if (childPid > 0) {
749         kill(childPid, SIGTERM);
750         childPid = -1;
751     }
752     for (rp = running_list; rp != NULL; rp = rp->next) {
753         if (rp->state != build_running) {
754             continue;
755         }
756         if (rp->pid > 0) {
757             kill(rp->pid, SIGTERM);
758             rp->pid = -1;
759         }
760     }
761     if (getpid() == getpgrp()) {
762         bsd_signal(SIGTERM, SIG_IGN);
763         kill(-getpid(), SIGTERM);
764     }
765     /* Clean up all parallel children already finished */
766     /* Clean up all parallel/distributed children already finished */
767     finish_children(false);
768     /* Make sure the processes running under us terminate first */
770     while (wait((int *) NULL) != -1);
771     /* Delete the current targets unless they are precious */
772     if ((current_target != NULL) &&
773         current_target->is_member &&
774         ((member = get_prop(current_target->prop, member_prop)) != NULL)) {
775         current_target = member->body.member.library;
776     }
777     if (!do_not_exec_rule &&
778         !touch &&
779         !quest &&
780         (current_target != NULL) &&
781         !(current_target->stat.is_precious || all_precious)) {
783     /* BID_1030811 */
784     /* azv 16 Oct 95 */
785         current_target->stat.time = file_no_time;
787         if (exists(current_target) != file_doesnt_exist) {
788             (void) fprintf(stderr,
789                 "\n*** %s ",
790                 current_target->string_mb);

```

```

791         if (current_target->stat.is_dir) {
792             (void) fprintf(stderr,
793                 catgets(catd, 1, 168, "not remove
794                 current_target->string_mb);
795         } else if (unlink(current_target->string_mb) == 0) {
796             (void) fprintf(stderr,
797                 catgets(catd, 1, 169, "removed.\n
798                 current_target->string_mb);
799         } else {
800             (void) fprintf(stderr,
801                 catgets(catd, 1, 170, "could not
802                 current_target->string_mb,
803                 errmsg(errno));
804         }
805     }
806 }
807 for (rp = running_list; rp != NULL; rp = rp->next) {
808     if (rp->state != build_running) {
809         continue;
810     }
811     if (rp->target->is_member &&
812         ((member = get_prop(rp->target->prop, member_prop)) !=
813          NULL)) {
814         rp->target = member->body.member.library;
815     }
816     if (!do_not_exec_rule &&
817         !touch &&
818         !quest &&
819         !(rp->target->stat.is_precious || all_precious)) {
821         rp->target->stat.time = file_no_time;
822         if (exists(rp->target) != file_doesnt_exist) {
823             (void) fprintf(stderr,
824                 "\n*** %s ",
825                 rp->target->string_mb);
826             if (rp->target->stat.is_dir) {
827                 (void) fprintf(stderr,
828                     catgets(catd, 1, 171, "no
829                     rp->target->string_mb);
830             } else if (unlink(rp->target->string_mb) == 0) {
831                 (void) fprintf(stderr,
832                     catgets(catd, 1, 172, "re
833                     rp->target->string_mb);
834             } else {
835                 (void) fprintf(stderr,
836                     catgets(catd, 1, 173, "co
837                     rp->target->string_mb,
838                     errmsg(errno));
839             }
840         }
841     }
842 }

845 /* Have we locked .make.state or .nse_depinfo? */
846 if ((make_state_lockfile != NULL) && (make_state_locked)) {
847     unlink(make_state_lockfile);
848     make_state_lockfile = NULL;
849     make_state_locked = false;
850 }
851 /*
852  * Re-read .make.state file (it might be changed by recursive make)
853  */
854 check_state(NULL);

856 report_dir_enter_leave(false);

```

```

858         exit_status = 2;
859         exit(2);
860     }
    _____ unchanged_portion_omitted _____

1685 /*
1686  *   read_files_and_state(argc, argv)
1687  *
1688  *   Read the makefiles we care about and the environment
1689  *   Also read the = style command line options
1690  *
1691  *   Parameters:
1692  *       argc           You know what this is
1693  *       argv           You know what this is
1694  *
1695  *   Static variables used:
1696  *       env_wins       make -e, determines if env vars are RO
1697  *       ignore_default_mk make -r, determines if make.rules is read
1698  *       not_auto_depen dwight
1699  *
1700  *   Global variables used:
1701  *       default_target_to_build Set to first proper target from file
1702  *       do_not_exec_rule Set to false when makfile is made
1703  *       dot             The Name ".", used to read current dir
1704  *       empty_name     The Name "", use as macro value
1705  *       keep_state     Set if KEEP_STATE is in environment
1706  *       make_state     The Name ".make.state", used to read file
1707  *       makefile_type  Set to type of file being read
1708  *       makeflags     The Name "MAKEFLAGS", used to set macro value
1709  *       not_auto      dwight
1710  *       read_trace_level Checked to see if the reader should trace
1711  *       report_dependencies If -P is on we do not read .make.state
1712  *       trace_reader   Set if reader should trace
1713  *       virtual_root   The Name "VIRTUAL_ROOT", used to check value
1714  */
1715 static void
1716 read_files_and_state(int argc, char **argv)
1717 {
1718     wchar_t           buffer[1000];
1719     wchar_t           buffer_posix[1000];
1720     register char     ch;
1721     register char     *cp;
1722     Property          def_make_macro = NULL;
1723     Name              def_make_name;
1724     Name              default_makefile;
1725     String_rec        dest;
1726     wchar_t           destbuffer[STRING_BUFFER_LENGTH];
1727     register int      i;
1728     register int      j;
1729     Name              keep_state_name;
1730     int               length;
1731     Name              Makefile;
1732     register Property macro;
1733     struct stat       make_state_stat;
1734     Name              makefile_name;
1735     register int      makefile_next = 0;
1736     register Boolean  makefile_read = false;
1737     String_rec        makeflags_string;
1738     String_rec        makeflags_string_posix;
1739     String_rec *      makeflags_string_current;
1740     Name              makeflags_value_saved;
1741     register Name     name;
1742     Name              new_make_value;
1743     Boolean           save_do_not_exec_rule;
1744     Name              sdotMakefile;

```

```

1745     Name                sdotmakefile_name;
1746     static wchar_t      state_file_str;
1747     static char          state_file_str_mb[MAXPATHLEN];
1748     static struct _Name state_filename;
1749     Boolean              temp;
1750     char                 tmp_char;
1751     wchar_t              *tmp_wcs_buffer;
1752     register Name        value;
1753     ASCII_Dyn_Array      makeflags_and_macro;
1754     Boolean              is_xpg4;

1756 /*
1757  * Remember current mode. It may be changed after reading makefile
1758  * and we will have to correct MAKEFLAGS variable.
1759  */
1760     is_xpg4 = posix;

1762     MBSTOWCS(wcs_buffer, NOCATGETS("KEEP_STATE"));
1763     keep_state_name = GETNAME(wcs_buffer, FIND_LENGTH);
1764     MBSTOWCS(wcs_buffer, NOCATGETS("Makefile"));
1765     Makefile = GETNAME(wcs_buffer, FIND_LENGTH);
1766     MBSTOWCS(wcs_buffer, NOCATGETS("makefile"));
1767     makefile_name = GETNAME(wcs_buffer, FIND_LENGTH);
1768     MBSTOWCS(wcs_buffer, NOCATGETS("s.makefile"));
1769     sdotmakefile_name = GETNAME(wcs_buffer, FIND_LENGTH);
1770     MBSTOWCS(wcs_buffer, NOCATGETS("s.Makefile"));
1771     sdotMakefile = GETNAME(wcs_buffer, FIND_LENGTH);

1773 /*
1774  * Set flag if NSE is active
1775  */

1795 /*
1774  * initialize global dependency entry for .NOT_AUTO
1775  */
1776     not_auto_depen->next = NULL;
1777     not_auto_depen->name = not_auto;
1778     not_auto_depen->automatic = not_auto_depen->stale = false;

1780 /*
1781  * Read internal definitions and rules.
1782  */
1783     if (read_trace_level > 1) {
1784         trace_reader = true;
1785     }
1786     if (!ignore_default_mk) {
1787         if (svr4) {
1788             MBSTOWCS(wcs_buffer, NOCATGETS("svr4.make.rules"));
1789             default_makefile = GETNAME(wcs_buffer, FIND_LENGTH);
1790         } else {
1791             MBSTOWCS(wcs_buffer, NOCATGETS("make.rules"));
1792             default_makefile = GETNAME(wcs_buffer, FIND_LENGTH);
1793         }
1794         default_makefile->stat.is_file = true;

1796         (void) read_makefile(default_makefile,
1797                             true,
1798                             false,
1799                             true);
1800     }

1802 /*
1803  * If the user did not redefine the MAKE macro in the
1804  * default makefile (make.rules), then we'd like to
1805  * change the macro value of MAKE to be some form
1806  * of argv[0] for recursive MAKE builds.

```

```

1807     /*
1808     MBSTOWCS(wcs_buffer, NOCATGETS("MAKE"));
1809     def_make_name = GETNAME(wcs_buffer, wslen(wcs_buffer));
1810     def_make_macro = get_prop(def_make_name->prop, macro_prop);
1811     if ((def_make_macro != NULL) &&
1812         (IS_EQUAL(def_make_macro->body.macro.value->string_mb,
1813                 NOCATGETS("make")))) {
1814         MBSTOWCS(wcs_buffer, argv_zero_string);
1815         new_make_value = GETNAME(wcs_buffer, wslen(wcs_buffer));
1816         (void) SETVAR(def_make_name,
1817                     new_make_value,
1818                     false);
1819     }

1821     default_target_to_build = NULL;
1822     trace_reader = false;

1824 /*
1825  * Read environment args. Let file args which follow override unless
1826  * -e option seen. If -e option is not mentioned.
1827  */
1828     read_environment(env_wins);
1829     if (getvar(virtual_root)->hash.length == 0) {
1830         maybe_append_prop(virtual_root, macro_prop)
1831         ->body.macro.exported = true;
1832         MBSTOWCS(wcs_buffer, "/");
1833         (void) SETVAR(virtual_root,
1834                     GETNAME(wcs_buffer, FIND_LENGTH),
1835                     false);
1836     }

1838 /*
1839  * We now scan mf_argv and argv to see if we need to set
1840  * any of the DMake-added options/variables in MAKEFLAGS.
1841  */

1843     makeflags_and_macro.start = 0;
1844     makeflags_and_macro.size = 0;
1845     enter_argv_values(mf_argc, mf_argv, &makeflags_and_macro);
1846     enter_argv_values(argc, argv, &makeflags_and_macro);

1848 /*
1849  * Set MFLAGS and MAKEFLAGS
1850  */
1851     Before reading makefile we do not know exactly which mode
1852     (posix or not) is used. So prepare two MAKEFLAGS strings
1853     for both posix and solaris modes because they are different.
1854  */
1855     INIT_STRING_FROM_STACK(makeflags_string, buffer);
1856     INIT_STRING_FROM_STACK(makeflags_string_posix, buffer_posix);
1857     append_char((int) hyphen_char, &makeflags_string);
1858     append_char((int) hyphen_char, &makeflags_string_posix);

1860     switch (read_trace_level) {
1861     case 2:
1862         append_char('D', &makeflags_string);
1863         append_char('D', &makeflags_string_posix);
1864     case 1:
1865         append_char('D', &makeflags_string);
1866         append_char('D', &makeflags_string_posix);
1867     }
1868     switch (debug_level) {
1869     case 2:
1870         append_char('d', &makeflags_string);
1871         append_char('d', &makeflags_string_posix);
1872     case 1:

```

```

1873         append_char('d', &makeflags_string);
1874         append_char('d', &makeflags_string_posix);
1875     }
1876     if (env_wins) {
1877         append_char('e', &makeflags_string);
1878         append_char('e', &makeflags_string_posix);
1879     }
1880     if (ignore_errors_all) {
1881         append_char('i', &makeflags_string);
1882         append_char('i', &makeflags_string_posix);
1883     }
1884     if (continue_after_error) {
1885         if (stop_after_error_ever_seen) {
1886             append_char('S', &makeflags_string_posix);
1887             append_char((int) space_char, &makeflags_string_posix);
1888             append_char((int) hyphen_char, &makeflags_string_posix);
1889         }
1890         append_char('k', &makeflags_string);
1891         append_char('k', &makeflags_string_posix);
1892     } else {
1893         if (stop_after_error_ever_seen
1894             && continue_after_error_ever_seen) {
1895             append_char('k', &makeflags_string_posix);
1896             append_char((int) space_char, &makeflags_string_posix);
1897             append_char((int) hyphen_char, &makeflags_string_posix);
1898             append_char('S', &makeflags_string_posix);
1899         }
1900     }
1901     if (do_not_exec_rule) {
1902         append_char('n', &makeflags_string);
1903         append_char('n', &makeflags_string_posix);
1904     }
1905     switch (report_dependencies_level) {
1906     case 4:
1907         append_char('P', &makeflags_string);
1908         append_char('P', &makeflags_string_posix);
1909     case 3:
1910         append_char('P', &makeflags_string);
1911         append_char('P', &makeflags_string_posix);
1912     case 2:
1913         append_char('P', &makeflags_string);
1914         append_char('P', &makeflags_string_posix);
1915     case 1:
1916         append_char('P', &makeflags_string);
1917         append_char('P', &makeflags_string_posix);
1918     }
1919     if (trace_status) {
1920         append_char('p', &makeflags_string);
1921         append_char('p', &makeflags_string_posix);
1922     }
1923     if (quest) {
1924         append_char('q', &makeflags_string);
1925         append_char('q', &makeflags_string_posix);
1926     }
1927     if (silent_all) {
1928         append_char('s', &makeflags_string);
1929         append_char('s', &makeflags_string_posix);
1930     }
1931     if (touch) {
1932         append_char('t', &makeflags_string);
1933         append_char('t', &makeflags_string_posix);
1934     }
1935     if (build_unconditional) {
1936         append_char('u', &makeflags_string);
1937         append_char('u', &makeflags_string_posix);
1938     }

```

```

1939     if (report_cwd) {
1940         append_char('w', &makeflags_string);
1941         append_char('w', &makeflags_string_posix);
1942     }
1943     /* -c dmake_rcfile */
1944     if (dmake_rcfile_specified) {
1945         MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_RCFILE"));
1946         dmake_rcfile = GETNAME(wcs_buffer, FIND_LENGTH);
1947         append_makeflags_string(dmake_rcfile, &makeflags_string);
1948         append_makeflags_string(dmake_rcfile, &makeflags_string_posix);
1949     }
1950     /* -g dmake_group */
1951     if (dmake_group_specified) {
1952         MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_GROUP"));
1953         dmake_group = GETNAME(wcs_buffer, FIND_LENGTH);
1954         append_makeflags_string(dmake_group, &makeflags_string);
1955         append_makeflags_string(dmake_group, &makeflags_string_posix);
1956     }
1957     /* -j dmake_max_jobs */
1958     if (dmake_max_jobs_specified) {
1959         MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_MAX_JOBS"));
1960         dmake_max_jobs = GETNAME(wcs_buffer, FIND_LENGTH);
1961         append_makeflags_string(dmake_max_jobs, &makeflags_string);
1962         append_makeflags_string(dmake_max_jobs, &makeflags_string_posix);
1963     }
1964     /* -m dmake_mode */
1965     if (dmake_mode_specified) {
1966         MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_MODE"));
1967         dmake_mode = GETNAME(wcs_buffer, FIND_LENGTH);
1968         append_makeflags_string(dmake_mode, &makeflags_string);
1969         append_makeflags_string(dmake_mode, &makeflags_string_posix);
1970     }
1971     /* -x dmake_compat_mode */
1972     // if (dmake_compat_mode_specified) {
1973     //     MBSTOWCS(wcs_buffer, NOCATGETS("SUN_MAKE_COMPAT_MODE"));
1974     //     dmake_compat_mode = GETNAME(wcs_buffer, FIND_LENGTH);
1975     //     append_makeflags_string(dmake_compat_mode, &makeflags_string);
1976     //     append_makeflags_string(dmake_compat_mode, &makeflags_string_posix);
1977     // }
1978     /* -x dmake_output_mode */
1979     if (dmake_output_mode_specified) {
1980         MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_OUTPUT_MODE"));
1981         dmake_output_mode = GETNAME(wcs_buffer, FIND_LENGTH);
1982         append_makeflags_string(dmake_output_mode, &makeflags_string);
1983         append_makeflags_string(dmake_output_mode, &makeflags_string_posix);
1984     }
1985     /* -o dmake_odir */
1986     if (dmake_odir_specified) {
1987         MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_ODIR"));
1988         dmake_odir = GETNAME(wcs_buffer, FIND_LENGTH);
1989         append_makeflags_string(dmake_odir, &makeflags_string);
1990         append_makeflags_string(dmake_odir, &makeflags_string_posix);
1991     }
1992     /* -M pmake_machinesfile */
1993     if (pmake_machinesfile_specified) {
1994         MBSTOWCS(wcs_buffer, NOCATGETS("PMAKE_MACHINESFILE"));
1995         pmake_machinesfile = GETNAME(wcs_buffer, FIND_LENGTH);
1996         append_makeflags_string(pmake_machinesfile, &makeflags_string);
1997         append_makeflags_string(pmake_machinesfile, &makeflags_string_posix);
1998     }
1999     /* -R */
2000     if (pmake_cap_r_specified) {
2001         append_char((int) space_char, &makeflags_string);
2002         append_char((int) hyphen_char, &makeflags_string);
2003         append_char('R', &makeflags_string);
2004         append_char((int) space_char, &makeflags_string_posix);

```

```

2005         append_char((int) hyphen_char, &makeflags_string_posix);
2006         append_char('R', &makeflags_string_posix);
2007     }

2009 /*
2010 *      Make sure MAKEFLAGS is exported
2011 */
2012 maybe_append_prop(makeflags, macro_prop)->
2013     body.macro.exported = true;

2015     if (makeflags_string.buffer.start[1] != (int) nul_char) {
2016         if (makeflags_string.buffer.start[1] != (int) space_char) {
2017             MBSTOWCS(wcs_buffer, NOCATGETS("MFLAGS"));
2018             (void) SETVAR(GETNAME(wcs_buffer, FIND_LENGTH),
2019                 GETNAME(makeflags_string.buffer.start,
2020                     FIND_LENGTH),
2021                 false);
2022         } else {
2023             MBSTOWCS(wcs_buffer, NOCATGETS("MFLAGS"));
2024             (void) SETVAR(GETNAME(wcs_buffer, FIND_LENGTH),
2025                 GETNAME(makeflags_string.buffer.start + 2,
2026                     FIND_LENGTH),
2027                 false);
2028         }
2029     }

2031 /*
2032 *      Add command line macro to POSIX makeflags_string
2033 */
2034     if (makeflags_and_macro.start) {
2035         tmp_char = (char) space_char;
2036         cp = makeflags_and_macro.start;
2037         do {
2038             append_char(tmp_char, &makeflags_string_posix);
2039         } while (tmp_char = *cp++ );
2040         retmem_mb(makeflags_and_macro.start);
2041     }

2043 /*
2044 *      Now set the value of MAKEFLAGS macro in accordance
2045 *      with current mode.
2046 */
2047     macro = maybe_append_prop(makeflags, macro_prop);
2048     temp = (Boolean) macro->body.macro.read_only;
2049     macro->body.macro.read_only = false;
2050     if (posix || gnu_style) {
2051         makeflags_string_current = &makeflags_string_posix;
2052     } else {
2053         makeflags_string_current = &makeflags_string;
2054     }
2055     if (makeflags_string_current->buffer.start[1] == (int) nul_char) {
2056         makeflags_value_saved =
2057             GETNAME( makeflags_string_current->buffer.start + 1
2058                 , FIND_LENGTH
2059                 );
2060     } else {
2061         if (makeflags_string_current->buffer.start[1] != (int) space_cha
2062             makeflags_value_saved =
2063                 GETNAME( makeflags_string_current->buffer.start
2064                     , FIND_LENGTH
2065                     );
2066     } else {
2067         makeflags_value_saved =
2068             GETNAME( makeflags_string_current->buffer.start
2069                 , FIND_LENGTH
2070                 );

```

```

2071     }
2072 }
2073 (void) SETVAR( makeflags
2074     , makeflags_value_saved
2075     , false
2076     );
2077 macro->body.macro.read_only = temp;

2079 /*
2080 *      Read command line "-f" arguments and ignore -c, g, j, K, M, m, O and o a
2081 */
2082 save_do_not_exec_rule = do_not_exec_rule;
2083 do_not_exec_rule = false;
2084 if (read_trace_level > 0) {
2085     trace_reader = true;
2086 }

2088 for (i = 1; i < argc; i++) {
2089     if (argv[i] &&
2090         (argv[i][0] == (int) hyphen_char) &&
2091         (argv[i][1] == 'f') &&
2092         (argv[i][2] == (int) nul_char)) {
2093         argv[i] = NULL; /* Remove -f */
2094         if (i >= argc - 1) {
2095             fatal(catgets(catd, 1, 190, "No filename argumen
2096         )
2097         MBSTOWCS(wcs_buffer, argv[++i]);
2098         primary_makefile = GETNAME(wcs_buffer, FIND_LENGTH);
2099         (void) read_makefile(primary_makefile, true, true, true)
2100         argv[i] = NULL; /* Remove filename */
2101         makefile_read = true;
2102     } else if (argv[i] &&
2103         (argv[i][0] == (int) hyphen_char) &&
2104         (argv[i][1] == 'c'
2105         || argv[i][1] == 'g'
2106         || argv[i][1] == 'j'
2107         || argv[i][1] == 'K'
2108         || argv[i][1] == 'M'
2109         || argv[i][1] == 'm'
2110         || argv[i][1] == 'O'
2111         || argv[i][1] == 'o') &&
2112         (argv[i][2] == (int) nul_char)) {
2113         argv[i] = NULL;
2114         argv[++i] = NULL;
2115     }
2116 }

2118 /*
2119 *      If no command line "-f" args then look for "makefile", and then for
2120 *      "Makefile" if "makefile" isn't found.
2121 */
2122 if (!makefile_read) {
2123     (void) read_dir(dot,
2124         (wchar_t *) NULL,
2125         (Property) NULL,
2126         (wchar_t *) NULL);
2127     if (!posix) {
2128         if (makefile_name->stat.is_file) {
2129             if (Makefile->stat.is_file) {
2130                 warning(catgets(catd, 1, 310, "Both 'makefile' a
2131             )
2132         }
2133         primary_makefile = makefile_name;
2134         makefile_read = read_makefile(makefile_name,
2135             false,
2136             false,
2137             true);

```



```

2137     }
2138     if (!makefile_read &&
2139         Makefile->stat.is_file) {
2140         primary_makefile = Makefile;
2141         makefile_read = read_makefile(Makefile,
2142                                     false,
2143                                     false,
2144                                     true);
2145     }
2146 } else {
2148     enum sccs_stat save_m_has_sccs = NO_SCCS;
2149     enum sccs_stat save_M_has_sccs = NO_SCCS;
2151
2152     if (makefile_name->stat.is_file) {
2153         if (Makefile->stat.is_file) {
2154             warning(catgets(catd, 1, 191, "Both 'makefile' a
2155         )
2156     }
2157     if (makefile_name->stat.is_file) {
2158         if (makefile_name->stat.has_sccs == NO_SCCS) {
2159             primary_makefile = makefile_name;
2160             makefile_read = read_makefile(makefile_name,
2161                                         false,
2162                                         false,
2163                                         true);
2164         } else {
2165             save_m_has_sccs = makefile_name->stat.has_sccs;
2166             makefile_name->stat.has_sccs = NO_SCCS;
2167             primary_makefile = makefile_name;
2168             makefile_read = read_makefile(makefile_name,
2169                                         false,
2170                                         false,
2171                                         true);
2172         }
2173     }
2174     if (!makefile_read &&
2175         Makefile->stat.is_file) {
2176         if (Makefile->stat.has_sccs == NO_SCCS) {
2177             primary_makefile = Makefile;
2178             makefile_read = read_makefile(Makefile,
2179                                         false,
2180                                         false,
2181                                         true);
2182         } else {
2183             save_M_has_sccs = Makefile->stat.has_sccs;
2184             Makefile->stat.has_sccs = NO_SCCS;
2185             primary_makefile = Makefile;
2186             makefile_read = read_makefile(Makefile,
2187                                         false,
2188                                         false,
2189                                         true);
2190         }
2191     }
2192     if (!makefile_read &&
2193         makefile_name->stat.is_file) {
2194         makefile_name->stat.has_sccs = save_m_has_sccs;
2195         primary_makefile = makefile_name;
2196         makefile_read = read_makefile(makefile_name,
2197                                     false,
2198                                     false,
2199                                     true);
2200     }
2201     if (!makefile_read &&
2202         Makefile->stat.is_file) {
2203         Makefile->stat.has_sccs = save_M_has_sccs;

```

```

2203         primary_makefile = Makefile;
2204         makefile_read = read_makefile(Makefile,
2205                                     false,
2206                                     false,
2207                                     true);
2208     }
2209 }
2210 }
2211 do_not_exec_rule = save_do_not_exec_rule;
2212 allrules_read = makefile_read;
2213 trace_reader = false;
2215 /*
2216 * Now get current value of MAKEFLAGS and compare it with
2217 * the saved value we set before reading makefile.
2218 * If they are different then MAKEFLAGS is subsequently set by
2219 * makefile, just leave it there. Otherwise, if make mode
2220 * is changed by using .POSIX target in makefile we need
2221 * to correct MAKEFLAGS value.
2222 */
2223 Name mf_val = getvar(makeflags);
2224 if ( (posix != is_xpg4)
2225     && (!strcmp(mf_val->string_mb, makeflags_value_saved->string_mb)))
2226 {
2227     if (makeflags_string_posix.buffer.start[1] == (int) nul_char) {
2228         (void) SETVAR(makeflags,
2229                     GETNAME(makeflags_string_posix.buffer.star
2230                             FIND_LENGTH),
2231                     false);
2232     } else {
2233         if (makeflags_string_posix.buffer.start[1] != (int) spac
2234             (void) SETVAR(makeflags,
2235                         GETNAME(makeflags_string_posix.buf
2236                                 FIND_LENGTH),
2237                         false);
2238     } else {
2239         (void) SETVAR(makeflags,
2240                     GETNAME(makeflags_string_posix.buf
2241                             FIND_LENGTH),
2242                     false);
2243     }
2244 }
2245 }
2247 if (makeflags_string.free_after_use) {
2248     retmem(makeflags_string.buffer.start);
2249 }
2250 if (makeflags_string_posix.free_after_use) {
2251     retmem(makeflags_string_posix.buffer.start);
2252 }
2253 makeflags_string.buffer.start = NULL;
2254 makeflags_string_posix.buffer.start = NULL;
2256 if (posix) {
2257     /*
2258     * If the user did not redefine the ARFLAGS macro in the
2259     * default makefile (make.rules), then we'd like to
2260     * change the macro value of ARFLAGS to be in accordance
2261     * with "POSIX" requirements.
2262     */
2263     MBSTOWCS(wcs_buffer, NOCATGETS("ARFLAGS"));
2264     name = GETNAME(wcs_buffer, wslen(wcs_buffer));
2265     macro = get_prop(name->prop, macro_prop);
2266     if ((macro != NULL) && /* Maybe (macro == NULL) || ? */
2267         (IS_EQUAL(macro->body.macro.value->string_mb,
2268                 NOCATGETS("rv")))) {

```

```

2269         MBSTOWCS(wcs_buffer, NOCATGETS("-rv"));
2270         value = GETNAME(wcs_buffer, wslen(wcs_buffer));
2271         (void) SETVAR(name,
2272                     value,
2273                     false);
2274     }
2275 }

2277 if (!posix && !svr4) {
2278     set_sgs_support();
2279 }

2282 /*
2283  * Make sure KEEP_STATE is in the environment if KEEP_STATE is on.
2284  */
2285 macro = get_prop(keep_state_name->prop, macro_prop);
2286 if ((macro != NULL) &&
2287     macro->body.macro.exported) {
2288     keep_state = true;
2289 }
2290 if (keep_state) {
2291     if (macro == NULL) {
2292         macro = maybe_append_prop(keep_state_name,
2293                                 macro_prop);
2294     }
2295     macro->body.macro.exported = true;
2296     (void) SETVAR(keep_state_name,
2297                 empty_name,
2298                 false);

2300     /*
2301      * Read state file
2302     */

2304     /* Before we read state, let's make sure we have
2305      ** right state file.
2306     */
2307     /* just in case macro references are used in make_state file
2308      ** name, we better expand them at this stage using expand_value.
2309     */
2310     INIT_STRING_FROM_STACK(dest, destbuffer);
2311     expand_value(make_state, &dest, false);

2313     make_state = GETNAME(dest.buffer.start, FIND_LENGTH);

2315     if(!stat(make_state->string_mb, &make_state_stat)) {
2316         if(!(make_state_stat.st_mode & S_IFREG) ) {
2317             /* copy the make_state structure to the other
2318              ** and then let make_state point to the new
2319              ** one.
2320             */
2321             memcpy(&state_filename, make_state, sizeof(state_filename))
2322             state_filename.string_mb = state_file_str_mb;
2323         /* Just a kludge to avoid two slashes back to back */
2324         if((make_state->hash.length == 1)&&
2325            (make_state->string_mb[0] == '/')) {
2326             make_state->hash.length = 0;
2327             make_state->string_mb[0] = '\0';
2328         }
2329         sprintf(state_file_str_mb, NOCATGETS("%s%s"),
2330             make_state->string_mb, NOCATGETS("/.make.state"));
2331         make_state = &state_filename;
2332         /* adjust the length to reflect the appended string */
2333         make_state->hash.length += 12;
2334     }

```

```

2335     } else { /* the file doesn't exist or no permission */
2336         char tmp_path[MAXPATHLEN];
2337         char *slashp;

2339         if (slashp = strrchr(make_state->string_mb, '/')) {
2340             strncpy(tmp_path, make_state->string_mb,
2341                 (slashp - make_state->string_mb));
2342             tmp_path[slashp - make_state->string_mb]=0;
2343             if(strlen(tmp_path)) {
2344                 if(stat(tmp_path, &make_state_stat)) {
2345                     warning(catgets(catd, 1, 192, "directory %s for .KEEP_
2346                 )
2347                 if (access(tmp_path, F_OK) != 0) {
2348                     warning(catgets(catd, 1, 193, "can't access dir %s"),t
2349                 )
2350             }
2351         }
2352     }
2353     if (report_dependencies_level != 1) {
2354         Makefile_type makefile_type_temp = makefile_type;
2355         makefile_type = reading_statefile;
2356         if (read_trace_level > 1) {
2357             trace_reader = true;
2358         }
2359         (void) read_simple_file(make_state,
2360                               false,
2361                               false,
2362                               false,
2363                               false,
2364                               false,
2365                               true);
2366         trace_reader = false;
2367         makefile_type = makefile_type_temp;
2368     }
2369 }
2370 }

```

unchanged_portion_omitted

```

*****
46136 Wed May 20 12:04:51 2015
new/usr/src/cmd/make/bin/parallel.cc
make: remove more distributed mode code
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright 2005 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

27 /*
28 *      parallel.cc
29 *
30 *      Deal with the parallel processing
31 */

33 /*
34 * Included files
35 */
36 #include <errno.h>          /* errno */
37 #include <fcntl.h>
38 #include <mk/defs.h>
39 #include <mksh/dosys.h>     /* redirect_io() */
40 #include <mksh/macro.h>     /* expand_value() */
41 #include <mksh/misc.h>      /* getmem() */
42 #include <sys/signal.h>
43 #include <sys/stat.h>
44 #include <sys/types.h>
45 #include <sys/utsname.h>
46 #include <sys/wait.h>
47 #include <unistd.h>
48 #include <netdb.h>

52 /*
53 * Defined macros
54 */
55 #define MAXRULES          100

57 /*
58 * This const should be in avo_dms/include/AvoDmakeCommand.h
59 */
60 const int local_host_mask = 0x20;

```

```

63 /*
64 * typedefs & structs
65 */

68 /*
69 * Static variables
70 */
71 static Boolean      just_did_subtree = false;
72 static char         local_host[MAXNAMELEN] = "";
73 static char         user_name[MAXNAMELEN] = "";
74 static int          pmake_max_jobs = 0;
75 static pid_t        process_running = -1;
76 static Running     *running_tail = &running_list;
77 static Name         subtree_conflict;
78 static Name         subtree_conflict2;

81 /*
82 * File table of contents
83 */
84 static void         delete_running_struct(Running rp);
85 static Boolean     dependency_conflict(Name target);
86 static Doname      distribute_process(char **commands, Property line);
87 static void        doname_subtree(Name target, Boolean do_get, Boolean impl);
88 static void        dump_out_file(char *filename, Boolean err);
89 static void        finish_doname(Running rp);
90 static void        maybe_reread_make_state(void);
91 static void        process_next(void);
92 static void        reset_conditionals(int cnt, Name *targets, Property *loc);
93 static pid_t       run_rule_commands(char *host, char **commands);
94 static Property    *set_conditionals(int cnt, Name *targets);
95 static void        store_conditionals(Running rp);

98 /*
99 *      execute_parallel(line, waitflg)
100 *
101 *      DMake 2.x:
102 *      parallel mode: spawns a parallel process to execute the command group.
103 *      distributed mode: sends the command group down the pipe to rxm.
104 *
105 *      Return value:
106 *
107 *      Parameters:
108 *          line          The command group to execute
109 */
110 Doname
111 execute_parallel(Property line, Boolean waitflg, Boolean local)
112 {
113     int          argcnt;
114     int          cmd_options = 0;
115     char         *commands[MAXRULES + 5];
116     char         *cp;
117     Name         dmake_name;
118     Name         dmake_value;
119     int          ignore;
120     Name         make_machines_name;
121     char         **p;
122     Property     prop;
123     Doname      result = build_ok;
124     Cmd_line    rule;
125     Boolean     silent_flag;
126     Name        target = line->body.line.target;

```

```

127     Boolean                wrote_state_file = false;
129     if ((pmake_max_jobs == 0) &&
130         (dmake_mode_type == parallel_mode)) {
131         if (local_host[0] == '\0') {
132             (void) gethostname(local_host, MAXNAMELEN);
133         }
134         MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_MAX_JOBS"));
135         dmake_name = GETNAME(wcs_buffer, FIND_LENGTH);
136         if (((prop = get_prop(dmake_name->prop, macro_prop)) != NULL) &&
137             ((dmake_value = prop->body.macro.value) != NULL)) {
138             pmake_max_jobs = atoi(dmake_value->string_mb);
139             if (pmake_max_jobs <= 0) {
140                 warning(catgets(catd, 1, 308, "DMAKE_MAX_JOBS ca
141                 warning(catgets(catd, 1, 309, "setting DMAKE_MAX
142                 pmake_max_jobs = PMAKE_DEF_MAX_JOBS;
143             }
144         } else {
145             /*
146             * For backwards compatibility w/ PMake 1.x, when
147             * DMake 2.x is being run in parallel mode, DMake
148             * should parse the PMake startup file
149             * $(HOME)/.make.machines to get the pmake_max_jobs.
150             */
151             MBSTOWCS(wcs_buffer, NOCATGETS("PMAKE_MACHINESFILE"));
152             dmake_name = GETNAME(wcs_buffer, FIND_LENGTH);
153             if (((prop = get_prop(dmake_name->prop, macro_prop)) !=
154                 ((dmake_value = prop->body.macro.value) != NULL)) {
155                 make_machines_name = dmake_value;
156             } else {
157                 make_machines_name = NULL;
158             }
159             if ((pmake_max_jobs = read_make_machines(make_machines_n
160                 pmake_max_jobs = PMAKE_DEF_MAX_JOBS;
161             )
162         }
163     }

165     if ((dmake_mode_type == serial_mode) ||
166         ((dmake_mode_type == parallel_mode) && (waitflg))) {
167         return (execute_serial(line));
168     }

170     {
171         p = commands;
172     }

174     argcnt = 0;
175     for (rule = line->body.line.command_used;
176         rule != NULL;
177         rule = rule->next) {
178         if (posix && (touch || quest) && !rule->always_exec) {
179             continue;
180         }
181         if (vpath_defined) {
182             rule->command_line =
183             vpath_translation(rule->command_line);
184         }
185         if (dmake_mode_type == distributed_mode) {
186             cmd_options = 0;
187             if (local) {
188                 cmd_options |= local_host_mask;
189             }
190         } else {
191             silent_flag = false;
186

```

```

187         ignore = 0;
188     }
189     if (rule->command_line->hash.length > 0) {
190         if (++argcnt == MAXRULES) {
191             if (dmake_mode_type == distributed_mode) {
192                 /* XXX - tell rxm to execute on local ho
193                 /* I WAS HERE!!! */
194             } else {
195                 /* Too many rules, run serially instead.
196                 return build_serial;
197             }
198         }
199     }
200     if (rule->silent && !silent) {
201         silent_flag = true;
202     }
203     if (rule->ignore_error) {
204         ignore++;
205     }
206     /* XXX - need to add support for + prefix */
207     if (silent_flag || ignore) {
208         *p = getmem((silent_flag ? 1 : 0) +
209                 ignore +
210                 (strlen(rule->
211                     command_line->
212                     string_mb)) +
213                 1);
214         cp = *p++;
215         if (silent_flag) {
216             *cp++ = (int) at_char;
217         }
218         if (ignore) {
219             *cp++ = (int) hyphen_char;
220         }
221         (void) strcpy(cp, rule->command_line->st
222     } else {
223         *p++ = rule->command_line->string_mb;
224     }
225 }
226 }
227 }

229     if ((argcnt == 0) ||
230         (report_dependencies_level > 0)) {
231         return build_ok;
232     }
233     *p = NULL;
234 }
235 Doname res = distribute_process(commands, line);
236 if (res == build_running) {
237     parallel_process_cnt++;
238 }
239 /*
240 * Return only those memory that were specially allocated
241 * for part of commands.
242 */
243 for (int i = 0; commands[i] != NULL; i++) {
244     if ((commands[i][0] == (int) at_char) ||
245         (commands[i][0] == (int) hyphen_char)) {
246         retmem_mb(commands[i]);
247     }
248 }
249 return res;
250 }

```

```

246 }
    unchanged_portion_omitted_

1168 /*
1169 *   finish_children(docheck)
1170 *
1171 *   Finishes the processing for all targets which were running
1172 *   and have now completed.
1173 *
1174 *   Parameters:
1175 *       docheck           Completely check the finished target
1176 *
1177 *   Static variables used:
1178 *       running_tail     The tail of the running list
1179 *
1180 *   Global variables used:
1181 *       continue_after_error -k flag
1182 *       fatal_in_progress True if we are finishing up after fatal err
1183 *       running_list     List of running processes
1184 */
1185 void
1186 finish_children(Boolean docheck)
1187 {
1188     int             cmds_length;
1189     Property        line;
1190     Property        line2;
1191     struct stat     out_buf;
1192     Running         rp;
1193     Running         *rp_prev;
1194     Cmd_line        rule;
1195     Boolean         silent_flag;

1197     for (rp_prev = &running_list, rp = running_list;
1198         rp != NULL;
1199         rp = rp->next) {
1200 bypass_for_loop_inc_4:
1201         /*
1202          * If the state is ok or failed, then this target has
1203          * finished building.
1204          * In parallel_mode, output the accumulated stdout/stderr.
1205          * Read the auto dependency stuff, handle a failed build,
1206          * update the target, then finish the doname process for
1207          * that target.
1208          */
1209         if (rp->state == build_ok || rp->state == build_failed) {
1210             *rp_prev = rp->next;
1211             if (rp->next == NULL) {
1212                 running_tail = rp_prev;
1213             }
1214             if ((line2 = rp->command) == NULL) {
1215                 line2 = get_prop(rp->target->prop, line_prop);
1216             }

1229             if (dmake_mode_type == distributed_mode) {
1230                 if (rp->make_refd) {
1231                     maybe_reread_make_state();
1232                 }
1233             } else {
1234                 /*
1235                  * Check if there were any job output
1236                  * from the parallel build.
1237                  */
1238                 if (rp->stdout_file != NULL) {
1239                     if (stat(rp->stdout_file, &out_buf) < 0) {
1240                         fatal(catgets(catd, 1, 130, "stat of %s

```

```

1226         rp->stdout_file,
1227         errmsg(errno));
1228     }

1230 #endif /* !codereview */
1231     if ((line2 != NULL) &&
1232         (out_buf.st_size > 0)) {
1233         cmds_length = 0;
1234         for (rule = line2->body.line.command_use
1235             silent_flag = silent;
1236             rule != NULL;
1237             rule = rule->next) {
1238             cmds_length += rule->command_lin
1239             silent_flag = BOOLEAN(silent fla
1240         }
1241         if (out_buf.st_size != cmds_length || si
1242             output_mode == txt2_mode) {
1243             dump_out_file(rp->stdout_file, f
1244         }
1245     }
1246     (void) unlink(rp->stdout_file);
1247     retmem_mb(rp->stdout_file);
1248     rp->stdout_file = NULL;
1249 }

1251 if (!out_err_same && (rp->stderr_file != NULL)) {
1252     if (stat(rp->stderr_file, &out_buf) < 0) {
1253         fatal(catgets(catd, 1, 130, "stat of %s
1254             rp->stderr_file,
1255             errmsg(errno));
1256     }
1257     if ((line2 != NULL) &&
1258         (out_buf.st_size > 0)) {
1259         dump_out_file(rp->stderr_file, true);
1260     }
1261     (void) unlink(rp->stderr_file);
1262     retmem_mb(rp->stderr_file);
1263     rp->stderr_file = NULL;
1264 }
1265 }
1266 check_state(rp->temp_file);
1267 if (rp->temp_file != NULL) {
1268     free_name(rp->temp_file);
1269 }
1270 rp->temp_file = NULL;
1271 if (rp->state == build_failed) {
1272     line = get_prop(rp->target->prop, line_prop);
1273     if (line != NULL) {
1274         line->body.line.command_used = NULL;
1275     }
1276     if (continue_after_error ||
1277         fatal_in_progress ||
1278         !docheck) {
1279         warning(catgets(catd, 1, 256, "Command f
1280             rp->command ? line2->body.line.t
1281             build_failed_seen = true;
1282     } else {
1283         /*
1284          * XXX??? - DMake needs to exit(),
1285          * but shouldn't call fatal().
1286          */
1287 #ifdef PRINT_EXIT_STATUS
1288         warning(NOCATGETS("I'm in finish_childre
1289 #endif

```

```
1291 fatal(catgets(catd, 1, 258, "Command fai
1292 rp->command ? line2->body.line.t
1293 )
1294 }
1295 if (!docheck) {
1296 delete_running_struct(rp);
1297 rp = *rp_prev;
1298 if (rp == NULL) {
1299 break;
1300 } else {
1301 goto bypass_for_loop_inc_4;
1302 }
1303 }
1304 update_target(get_prop(rp->target->prop, line_prop),
1305 rp->state);
1306 finish_doname(rp);
1307 delete_running_struct(rp);
1308 rp = *rp_prev;
1309 if (rp == NULL) {
1310 break;
1311 } else {
1312 goto bypass_for_loop_inc_4;
1313 }
1314 } else {
1315 rp_prev = &rp->next;
1316 }
1317 }
1318 }
_unchanged_portion_omitted_
```

new/usr/src/cmd/make/include/mk/defs.h

1

14080 Wed May 20 12:04:52 2015

new/usr/src/cmd/make/include/mk/defs.h

make: remove more distributed mode code

unchanged_portion_omitted_

```
132 typedef enum {
133     serial_mode,
134     parallel_mode
134     parallel_mode,
135     distributed_mode
135 } DMake_mode;
unchanged_portion_omitted_
```