```
**********************************************************
   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_

extern  Name            normalize_name(register wchar_t *name_string, register i

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

static  void            append_makeflags_string(Name, String);
static  void            doalarm(int);
static  void            enter_argv_values(int , char **, ASCII_Dyn_Array *);
static  void            make_targets(int, char **, Boolean);
static  int             parse_command_option(char);
static  void            read_command_options(int, char **);
static  void            read_environment(Boolean);
static  void            read_files_and_state(int, char **);
static  Boolean         read_makefile(Name, Boolean, Boolean, Boolean);
static  void            report_recursion(Name);
static  void            set_sgs_support(void);
static  void            setup_for_projectdir(void);
static  void            setup_makeflags_argv(void);
static  void            report_dir_enter_leave(Boolean entering);

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

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

jmp_buf jmpbuffer;
extern nl_catd catd;

/*
 *      main(argc, argv)
 *
 *      Parameters:
 *              argc                    You know what this is
 *              argv                    You know what this is
 *
 *      Static variables used:
 *              list_all_targets        make -T seen
 *              trace_status            make -p seen
 *
 *      Global variables used:
 *              debug_level             Should we trace make actions?
 *              keep_state              Set if .KEEP_STATE seen
 *              makeflags               The Name "MAKEFLAGS", used to get macro
 *              remote_command_name     Name of remote invocation cmd ("on")
 *              running_list            List of parallel running processes
 *              stdout_stderr_same      true if stdout and stderr are the same
 *              auto_dependencies       The Name "SUNPRO_DEPENDENCIES"
 *              temp_file_directory     Set to the dir where we create tmp file
 *              trace_reader            Set to reflect tracing status
 *              working_on_targets      Set when building user targets
 */
int
main(int argc, char *argv[])
{
        /*
         * cp is a -> to the value of the MAKEFLAGS env var,
         * which has to be regular chars.
         */
        register char           *cp;
        char                    make_state_dir[MAXPATHLEN];
        Boolean                 parallel_flag = false;
```

```
        char                    *prognameptr;
        char                    *slash_ptr;
        mode_t                  um;
        int                     i;
        struct itimerval        value;
        char                    def_dmakerc_path[MAXPATHLEN];
        Name                    dmake_name, dmake_name2;
        Name                    dmake_value, dmake_value2;
        Property                prop, prop2;
        struct stat             statbuf;
        int                     statval;

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

        (void) setlocale(LC_ALL, "");


#ifdef DMAKE_STATISTICS
        if (getenv(NOCATGETS("DMAKE_STATISTICS"))) {
                getname_stat = true;
        }
#endif

        catd = catopen(AVO_DOMAIN_DMAKE, NL_CAT_LOCALE);

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


/*
 * I put libmksdmsi18n_init() under #ifdef because it requires avo_i18n_init()
 * from avo_util library.
 */
        libmksdmsi18n_init();


        textdomain(NOCATGETS("SUNW_SPRO_MAKE"));

        g_argc = argc;
        g_argv = (char **) malloc((g_argc + 1) * sizeof(char *));
        for (i = 0; i < argc; i++) {
                g_argv[i] = argv[i];
        }
        g_argv[i] = NULL;

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

        if (*argv[0] == (int) slash_char) {
                /* argv[0] starts with a slash */
                argv_zero_string = strdup(argv[0]);
        } else if (strchr(argv[0], (int) slash_char) == NULL) {
                /* argv[0] contains no slashes */
                argv_zero_string = strdup(argv[0]);
        } else {
                /*
                 * argv[0] contains at least one slash,
                 * but doesn’t start with a slash
                 */
                char    *tmp_current_path;
                char    *tmp_string;

                tmp_current_path = get_current_path();
                tmp_string = getmem(strlen(tmp_current_path) + 1 +
                                    strlen(argv[0]) + 1);
                (void) sprintf(tmp_string,
                               "%s/%s",
                               tmp_current_path,
                               argv[0]);
                argv_zero_string = strdup(tmp_string);
                retmem_mb(tmp_string);
        }

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


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

        /* Sun OS make standart */
        svr4 = false;
        posix = false;
        if(!strcmp(argv_zero_string, NOCATGETS("/usr/xpg4/bin/make"))) {
                svr4 = false;
                posix = true;
        } else {
                prognameptr = strrchr(argv[0], '/');
                if(prognameptr) {
                        prognameptr++;
                } else {
                        prognameptr = argv[0];
                }
                if(!strcmp(prognameptr, NOCATGETS("svr4.make"))) {
                        svr4 = true;
                        posix = false;
                }
        }
        if (getenv(USE_SVR4_MAKE) || getenv(NOCATGETS("USE_SVID"))){
           svr4 = true;
           posix = false;
        }

        /*
         * Find the dmake_compat_mode: posix, sun, svr4, or gnu_style, .
         */
        char * dmake_compat_mode_var = getenv(NOCATGETS("SUN_MAKE_COMPAT_MODE"))
        if (dmake_compat_mode_var != NULL) {
                if (0 == strcasecmp(dmake_compat_mode_var, NOCATGETS("GNU"))) {
                        gnu_style = true;
                }
                //svr4 = false;
                //posix = false;
        }

        /*
         * Temporary directory set up.
         */
```

```
        char * tmpdir_var = getenv(NOCATGETS("TMPDIR"));
        if (tmpdir_var != NULL && *tmpdir_var == '/' && strlen(tmpdir_var) < MAX
                strcpy(mbs_buffer, tmpdir_var);
                for (tmpdir_var = mbs_buffer+strlen(mbs_buffer);
                        *(--tmpdir_var) == '/' && tmpdir_var > mbs_buffer;
                        *tmpdir_var = '\0');
                if (strlen(mbs_buffer) + 32 < MAXPATHLEN) { /* 32 = strlen("/dma
                        sprintf(mbs_buffer2, NOCATGETS("%s/dmake.tst.%d.XXXXXX")
                                mbs_buffer, getpid());
                        int fd = mkstemp(mbs_buffer2);
                        if (fd >= 0) {
                                close(fd);
                                unlink(mbs_buffer2);
                                tmpdir = strdup(mbs_buffer);
                        }
                }
        }

        /* find out if stdout and stderr point to the same place */
        if (fstat(1, &out_stat) < 0) {
                fatal(catgets(catd, 1, 165, "fstat of standard out failed: %s"),
        }
        if (fstat(2, &err_stat) < 0) {
                fatal(catgets(catd, 1, 166, "fstat of standard error failed: %s"
        }
        if ((out_stat.st_dev == err_stat.st_dev) &&
            (out_stat.st_ino == err_stat.st_ino)) {
                stdout_stderr_same = true;
        } else {
                stdout_stderr_same = false;
        }
        /* Make the vroot package scan the path using shell semantics */
        set_path_style(0);

        setup_char_semantics();

        setup_for_projectdir();

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

        load_cached_names();

/*
 *      Set command line flags
 */
        setup_makeflags_argv();
        read_command_options(mf_argc, mf_argv);
        read_command_options(argc, argv);
        if (debug_level > 0) {
                cp = getenv(makeflags->string_mb);
                (void) printf(catgets(catd, 1, 167, "MAKEFLAGS value: %s\n"), cp
        }

        setup_interrupt(handle_interrupt);

        read_files_and_state(argc, argv);

        /*
         * Find the dmake_output_mode: TXT1, TXT2 or HTML1.
         */
        MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_OUTPUT_MODE"));
        dmake_name2 = GETNAME(wcs_buffer, FIND_LENGTH);
        prop2 = get_prop(dmake_name2->prop, macro_prop);
        if (prop2 == NULL) {
                /* DMAKE_OUTPUT_MODE not defined, default to TXT1 mode */
                output_mode = txt1_mode;
        } else {
                dmake_value2 = prop2->body.macro.value;
                if ((dmake_value2 == NULL) ||
                    (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("TXT1")))) {
                        output_mode = txt1_mode;
                } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("TXT2")))
                        output_mode = txt2_mode;
                } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("HTML1"))
                        output_mode = html1_mode;
                } else {
                        warning(catgets(catd, 1, 352, "Unsupported value ‘%s’ fo
                              dmake_value2->string_mb);
                }
        }
        /*
         * Find the dmake_mode: parallel, or serial.
         * Find the dmake_mode: distributed, parallel, or serial.
         */
    if ((!pmake_cap_r_specified) &&
        (!pmake_machinesfile_specified)) {
        MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_MODE"));
        dmake_name2 = GETNAME(wcs_buffer, FIND_LENGTH);
        prop2 = get_prop(dmake_name2->prop, macro_prop);
        if (prop2 == NULL) {
                /* DMAKE_MODE not defined, default to parallel mode */
                dmake_mode_type = parallel_mode;
                /* DMAKE_MODE not defined, default to distributed mode */
                dmake_mode_type = distributed_mode;
                no_parallel = false;
        } else {
                dmake_value2 = prop2->body.macro.value;
                if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("parallel"))) {
                if ((dmake_value2 == NULL) ||
                    (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("distributed"))
                        dmake_mode_type = distributed_mode;
                        no_parallel = false;
                } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("parallel
                        dmake_mode_type = parallel_mode;
                        no_parallel = false;
                } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("serial")
                        dmake_mode_type = serial_mode;
                        no_parallel = true;
                } else {
                        fatal(catgets(catd, 1, 307, "Unknown dmake mode argument
                }
        }

        if ((!list_all_targets) &&
            (report_dependencies_level == 0)) {
                /*
                 * Check to see if either DMAKE_RCFILE or DMAKE_MODE is defined.
                 * They could be defined in the env, in the makefile, or on the
                 * command line.
                 * If neither is defined, and $(HOME)/.dmakerc does not exists,
                 * then print a message, and default to parallel mode.
                 */
                if(dmake_mode_type == distributed_mode) {
                        dmake_mode_type = parallel_mode;
                        no_parallel = false;
                }
        }
    }
```

```
        parallel_flag = true;
        putenv(strdup(NOCATGETS("DMAKE_CHILD=TRUE")));

//
// If dmake is running with -t option, set dmake_mode_type to serial.
// This is done because doname() calls touch_command() that runs serially.
// If we do not do that, maketool will have problems.
//
        if(touch) {
                dmake_mode_type = serial_mode;
                no_parallel = true;
        }

        /*
         * Check whether stdout and stderr are physically same.
         * This is in order to decide whether we need to redirect
         * stderr separately from stdout.
         * This check is performed only if __DMAKE_SEPARATE_STDERR
         * is not set. This variable may be used in order to preserve
         * the ’old’ behaviour.
         */
        out_err_same = true;
        char * dmake_sep_var = getenv(NOCATGETS("__DMAKE_SEPARATE_STDERR"));
        if (dmake_sep_var == NULL || (0 != strcasecmp(dmake_sep_var, NOCATGETS("
                struct stat stdout_stat;
                struct stat stderr_stat;
                if( (fstat(1, &stdout_stat) == 0)
                 && (fstat(2, &stderr_stat) == 0) )
                {
                        if( (stdout_stat.st_dev != stderr_stat.st_dev)
                         || (stdout_stat.st_ino != stderr_stat.st_ino) )
                        {
                                out_err_same = false;
                        }
                }
        }


/*
 *      Enable interrupt handler for alarms
 */
        (void) bsd_signal(SIGALRM, (SIG_PF)doalarm);

/*
 *      Check if make should report
 */
        if (getenv(sunpro_dependencies->string_mb) != NULL) {
                FILE    *report_file;

                report_dependency("");
                report_file = get_report_file();
                if ((report_file != NULL) && (report_file != (FILE*)-1)) {
                        (void) fprintf(report_file, "\n");
                }
        }

/*
 *      Make sure SUNPRO_DEPENDENCIES is exported (or not) properly.
 */
        if (keep_state) {
                maybe_append_prop(sunpro_dependencies, macro_prop)->
                  body.macro.exported = true;
        } else {
                maybe_append_prop(sunpro_dependencies, macro_prop)->
                  body.macro.exported = false;
        }

        working_on_targets = true;
        if (trace_status) {
                dump_make_state();
                fclose(stdout);
                fclose(stderr);
                exit_status = 0;
                exit(0);
        }
        if (list_all_targets) {
                dump_target_list();
                fclose(stdout);
                fclose(stderr);
                exit_status = 0;
                exit(0);
        }
        trace_reader = false;

        /*
         * Set temp_file_directory to the directory the .make.state
         * file is written to.
         */
        if ((slash_ptr = strrchr(make_state->string_mb, (int) slash_char)) == NU
                temp_file_directory = strdup(get_current_path());
        } else {
                *slash_ptr = (int) nul_char;
                (void) strcpy(make_state_dir, make_state->string_mb);
                *slash_ptr = (int) slash_char;
                   /* when there is only one slash and it’s the first
                   ** character, make_state_dir should point to ’/’.
                   */
                if(make_state_dir[0] == ’\0’) {
                   make_state_dir[0] = ’/’;
                   make_state_dir[1] = ’\0’;
                }
                if (make_state_dir[0] == (int) slash_char) {
                        temp_file_directory = strdup(make_state_dir);
                } else {
                        char    tmp_current_path2[MAXPATHLEN];

                        (void) sprintf(tmp_current_path2,
                                       "%s/%s",
                                       get_current_path(),
                                       make_state_dir);
                        temp_file_directory = strdup(tmp_current_path2);
                }
        }


        report_dir_enter_leave(true);

        make_targets(argc, argv, parallel_flag);

        report_dir_enter_leave(false);

        if (build_failed_ever_seen) {
                if (posix) {
                        exit_status = 1;
                }
                exit(1);
        }
        exit_status = 0;
        exit(0);
        /* NOTREACHED */
}
```

______***unchanged_portion_omitted_***

```
/*
 *      handle_interrupt()
 *
 *      This is where C-C traps are caught.
 *
 *      Parameters:
 *
 *      Global variables used (except DMake 1.0):
 *              current_target          Sometimes the current target is removed
 *              do_not_exec_rule        But not if -n is on
 *              quest                   or -q
 *              running_list            List of parallel running processes
 *              touch                   Current target is not removed if -t on
 */
void
handle_interrupt(int)
{
        Property                member;
        Running                 rp;

        (void) fflush(stdout);
        if (childPid > 0) {
                kill(childPid, SIGTERM);
                childPid = -1;
        }
        for (rp = running_list; rp != NULL; rp = rp->next) {
                if (rp->state != build_running) {
                        continue;
                }
                if (rp->pid > 0) {
                        kill(rp->pid, SIGTERM);
                        rp->pid = -1;
                }
        }
        if (getpid() == getpgrp()) {
                bsd_signal(SIGTERM, SIG_IGN);
                kill (-getpid(), SIGTERM);
        }
        /* Clean up all parallel children already finished */
        /* Clean up all parallel/distributed children already finished */
        finish_children(false);

        /* Make sure the processes running under us terminate first */

        while (wait((int *) NULL) != -1);
        /* Delete the current targets unless they are precious */
        if ((current_target != NULL) &&
            current_target->is_member &&
            ((member = get_prop(current_target->prop, member_prop)) != NULL)) {
                current_target = member->body.member.library;
        }
        if (!do_not_exec_rule &&
            !touch &&
            !quest &&
            (current_target != NULL) &&
            !(current_target->stat.is_precious || all_precious)) {

/* BID_1030811 */
/* azv 16 Oct 95 */
                current_target->stat.time = file_no_time;

                if (exists(current_target) != file_doesnt_exist) {
                        (void) fprintf(stderr,
                                       "\n*** %s ",
                                       current_target->string_mb);
                        if (current_target->stat.is_dir) {
                                (void) fprintf(stderr,
                                               catgets(catd, 1, 168, "not remove
                                               current_target->string_mb);
                        } else if (unlink(current_target->string_mb) == 0) {
                                (void) fprintf(stderr,
                                               catgets(catd, 1, 169, "removed.\n
                                               current_target->string_mb);
                        } else {
                                (void) fprintf(stderr,
                                               catgets(catd, 1, 170, "could not
                                               current_target->string_mb,
                                               errmsg(errno));
                        }
                }
        }
        for (rp = running_list; rp != NULL; rp = rp->next) {
                if (rp->state != build_running) {
                        continue;
                }
                if (rp->target->is_member &&
                    ((member = get_prop(rp->target->prop, member_prop)) !=
                     NULL)) {
                        rp->target = member->body.member.library;
                }
                if (!do_not_exec_rule &&
                    !touch &&
                    !quest &&
                    !(rp->target->stat.is_precious || all_precious)) {

                        rp->target->stat.time = file_no_time;
                        if (exists(rp->target) != file_doesnt_exist) {
                                (void) fprintf(stderr,
                                               "\n*** %s ",
                                               rp->target->string_mb);
                                if (rp->target->stat.is_dir) {
                                        (void) fprintf(stderr,
                                                       catgets(catd, 1, 171, "no
                                                       rp->target->string_mb);
                                } else if (unlink(rp->target->string_mb) == 0) {
                                        (void) fprintf(stderr,
                                                       catgets(catd, 1, 172, "re
                                                       rp->target->string_mb);
                                } else {
                                        (void) fprintf(stderr,
                                                       catgets(catd, 1, 173, "co
                                                       rp->target->string_mb,
                                                       errmsg(errno));
                                }
                        }
                }
        }


        /* Have we locked .make.state or .nse_depinfo? */
        if ((make_state_lockfile != NULL) && (make_state_locked)) {
                unlink(make_state_lockfile);
                make_state_lockfile = NULL;
                make_state_locked = false;
        }
        /*
         * Re-read .make.state file (it might be changed by recursive make)
         */
        check_state(NULL);

        report_dir_enter_leave(false);
```

```

        exit_status = 2;
        exit(2);
}
______unchanged_portion_omitted_

/*
 *      read_files_and_state(argc, argv)
 *
 *      Read the makefiles we care about and the environment
 *      Also read the = style command line options
 *
 *      Parameters:
 *              argc            You know what this is
 *              argv            You know what this is
 *
 *      Static variables used:
 *              env_wins        make -e, determines if env vars are RO
 *              ignore_default_mk make -r, determines if make.rules is read
 *              not_auto_depen  dwight
 *
 *      Global variables used:
 *              default_target_to_build Set to first proper target from file
 *              do_not_exec_rule Set to false when makfile is made
 *              dot             The Name ".", used to read current dir
 *              empty_name      The Name "", use as macro value
 *              keep_state      Set if KEEP_STATE is in environment
 *              make_state      The Name ".make.state", used to read file
 *              makefile_type   Set to type of file being read
 *              makeflags       The Name "MAKEFLAGS", used to set macro value
 *              not_auto        dwight
 *              read_trace_level Checked to se if the reader should trace
 *              report_dependencies If -P is on we do not read .make.state
 *              trace_reader    Set if reader should trace
 *              virtual_root    The Name "VIRTUAL_ROOT", used to check value
 */
static void
read_files_and_state(int argc, char **argv)
{
        wchar_t                 buffer[1000];
        wchar_t                 buffer_posix[1000];
        register char           ch;
        register char           *cp;
        Property                def_make_macro = NULL;
        Name                    def_make_name;
        Name                    default_makefile;
        String_rec              dest;
        wchar_t                 destbuffer[STRING_BUFFER_LENGTH];
        register int            i;
        register int            j;
        Name                    keep_state_name;
        int                     length;
        Name                    Makefile;
        register Property       macro;
        struct stat             make_state_stat;
        Name                    makefile_name;
        register int            makefile_next = 0;
        register Boolean        makefile_read = false;
        String_rec              makeflags_string;
        String_rec              makeflags_string_posix;
        String_rec *            makeflags_string_current;
        Name                    makeflags_value_saved;
        register Name           name;
        Name                    new_make_value;
        Boolean                 save_do_not_exec_rule;
        Name                    sdotMakefile;
        Name                    sdotmakefile_name;
        static wchar_t          state_file_str;
        static char             state_file_str_mb[MAXPATHLEN];
        static struct _Name     state_filename;
        Boolean                 temp;
        char                    tmp_char;
        wchar_t                 *tmp_wcs_buffer;
        register Name           value;
        ASCII_Dyn_Array         makeflags_and_macro;
        Boolean                 is_xpg4;

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

        MBSTOWCS(wcs_buffer, NOCATGETS("KEEP_STATE"));
        keep_state_name = GETNAME(wcs_buffer, FIND_LENGTH);
        MBSTOWCS(wcs_buffer, NOCATGETS("Makefile"));
        Makefile = GETNAME(wcs_buffer, FIND_LENGTH);
        MBSTOWCS(wcs_buffer, NOCATGETS("makefile"));
        makefile_name = GETNAME(wcs_buffer, FIND_LENGTH);
        MBSTOWCS(wcs_buffer, NOCATGETS("s.makefile"));
        sdotmakefile_name = GETNAME(wcs_buffer, FIND_LENGTH);
        MBSTOWCS(wcs_buffer, NOCATGETS("s.Makefile"));
        sdotMakefile = GETNAME(wcs_buffer, FIND_LENGTH);

/*
 *      Set flag if NSE is active
 */

/*
 *      initialize global dependency entry for .NOT_AUTO
 */
        not_auto_depen->next = NULL;
        not_auto_depen->name = not_auto;
        not_auto_depen->automatic = not_auto_depen->stale = false;

/*
 *      Read internal definitions and rules.
 */
        if (read_trace_level > 1) {
                trace_reader = true;
        }
        if (!ignore_default_mk) {
                if (svr4) {
                        MBSTOWCS(wcs_buffer, NOCATGETS("svr4.make.rules"));
                        default_makefile = GETNAME(wcs_buffer, FIND_LENGTH);
                } else {
                        MBSTOWCS(wcs_buffer, NOCATGETS("make.rules"));
                        default_makefile = GETNAME(wcs_buffer, FIND_LENGTH);
                }
                default_makefile->stat.is_file = true;

                (void) read_makefile(default_makefile,
                                     true,
                                     false,
                                     true);
        }

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

```
         */
        MBSTOWCS(wcs_buffer, NOCATGETS("MAKE"));
        def_make_name = GETNAME(wcs_buffer, wslen(wcs_buffer));
        def_make_macro = get_prop(def_make_name->prop, macro_prop);
        if ((def_make_macro != NULL) &&
            (IS_EQUAL(def_make_macro->body.macro.value->string_mb,
                      NOCATGETS("make")))) {
                MBSTOWCS(wcs_buffer, argv_zero_string);
                new_make_value = GETNAME(wcs_buffer, wslen(wcs_buffer));
                (void) SETVAR(def_make_name,
                              new_make_value,
                              false);
        }

        default_target_to_build = NULL;
        trace_reader = false;

/*
 *      Read environment args. Let file args which follow override unless
 *      -e option seen. If -e option is not mentioned.
 */
        read_environment(env_wins);
        if (getvar(virtual_root)->hash.length == 0) {
                maybe_append_prop(virtual_root, macro_prop)
                  ->body.macro.exported = true;
                MBSTOWCS(wcs_buffer, "/");
                (void) SETVAR(virtual_root,
                              GETNAME(wcs_buffer, FIND_LENGTH),
                              false);
        }

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

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

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

        switch (read_trace_level) {
        case 2:
                append_char('D', &makeflags_string);
                append_char('D', &makeflags_string_posix);
        case 1:
                append_char('D', &makeflags_string);
                append_char('D', &makeflags_string_posix);
        }
        switch (debug_level) {
        case 2:
                append_char('d', &makeflags_string);
                append_char('d', &makeflags_string_posix);
        case 1:
                append_char('d', &makeflags_string);
                append_char('d', &makeflags_string_posix);
        }
        if (env_wins) {
                append_char('e', &makeflags_string);
                append_char('e', &makeflags_string_posix);
        }
        if (ignore_errors_all) {
                append_char('i', &makeflags_string);
                append_char('i', &makeflags_string_posix);
        }
        if (continue_after_error) {
                if (stop_after_error_ever_seen) {
                        append_char('S', &makeflags_string_posix);
                        append_char((int) space_char, &makeflags_string_posix);
                        append_char((int) hyphen_char, &makeflags_string_posix);
                }
                append_char('k', &makeflags_string);
                append_char('k', &makeflags_string_posix);
        } else {
                if (stop_after_error_ever_seen
                    && continue_after_error_ever_seen) {
                        append_char('k', &makeflags_string_posix);
                        append_char((int) space_char, &makeflags_string_posix);
                        append_char((int) hyphen_char, &makeflags_string_posix);
                        append_char('S', &makeflags_string_posix);
                }
        }
        if (do_not_exec_rule) {
                append_char('n', &makeflags_string);
                append_char('n', &makeflags_string_posix);
        }
        switch (report_dependencies_level) {
        case 4:
                append_char('P', &makeflags_string);
                append_char('P', &makeflags_string_posix);
        case 3:
                append_char('P', &makeflags_string);
                append_char('P', &makeflags_string_posix);
        case 2:
                append_char('P', &makeflags_string);
                append_char('P', &makeflags_string_posix);
        case 1:
                append_char('P', &makeflags_string);
                append_char('P', &makeflags_string_posix);
        }
        if (trace_status) {
                append_char('p', &makeflags_string);
                append_char('p', &makeflags_string_posix);
        }
        if (quest) {
                append_char('q', &makeflags_string);
                append_char('q', &makeflags_string_posix);
        }
        if (silent_all) {
                append_char('s', &makeflags_string);
                append_char('s', &makeflags_string_posix);
        }
        if (touch) {
                append_char('t', &makeflags_string);
                append_char('t', &makeflags_string_posix);
        }
        if (build_unconditional) {
                append_char('u', &makeflags_string);
                append_char('u', &makeflags_string_posix);
        }
```

```
        if (report_cwd) {
                append_char('w', &makeflags_string);
                append_char('w', &makeflags_string_posix);
        }
        /* -c dmake_rcfile */
        if (dmake_rcfile_specified) {
                MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_RCFILE"));
                dmake_rcfile = GETNAME(wcs_buffer, FIND_LENGTH);
                append_makeflags_string(dmake_rcfile, &makeflags_string);
                append_makeflags_string(dmake_rcfile, &makeflags_string_posix);
        }
        /* -g dmake_group */
        if (dmake_group_specified) {
                MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_GROUP"));
                dmake_group = GETNAME(wcs_buffer, FIND_LENGTH);
                append_makeflags_string(dmake_group, &makeflags_string);
                append_makeflags_string(dmake_group, &makeflags_string_posix);
        }
        /* -j dmake_max_jobs */
        if (dmake_max_jobs_specified) {
                MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_MAX_JOBS"));
                dmake_max_jobs = GETNAME(wcs_buffer, FIND_LENGTH);
                append_makeflags_string(dmake_max_jobs, &makeflags_string);
                append_makeflags_string(dmake_max_jobs, &makeflags_string_posix)
        }
        /* -m dmake_mode */
        if (dmake_mode_specified) {
                MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_MODE"));
                dmake_mode = GETNAME(wcs_buffer, FIND_LENGTH);
                append_makeflags_string(dmake_mode, &makeflags_string);
                append_makeflags_string(dmake_mode, &makeflags_string_posix);
        }
        /* -x dmake_compat_mode */
//      if (dmake_compat_mode_specified) {
//              MBSTOWCS(wcs_buffer, NOCATGETS("SUN_MAKE_COMPAT_MODE"));
//              dmake_compat_mode = GETNAME(wcs_buffer, FIND_LENGTH);
//              append_makeflags_string(dmake_compat_mode, &makeflags_string);
//              append_makeflags_string(dmake_compat_mode, &makeflags_string_pos
//      }
        /* -x dmake_output_mode */
        if (dmake_output_mode_specified) {
                MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_OUTPUT_MODE"));
                dmake_output_mode = GETNAME(wcs_buffer, FIND_LENGTH);
                append_makeflags_string(dmake_output_mode, &makeflags_string);
                append_makeflags_string(dmake_output_mode, &makeflags_string_pos
        }
        /* -o dmake_odir */
        if (dmake_odir_specified) {
                MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_ODIR"));
                dmake_odir = GETNAME(wcs_buffer, FIND_LENGTH);
                append_makeflags_string(dmake_odir, &makeflags_string);
                append_makeflags_string(dmake_odir, &makeflags_string_posix);
        }
        /* -M pmake_machinesfile */
        if (pmake_machinesfile_specified) {
                MBSTOWCS(wcs_buffer, NOCATGETS("PMAKE_MACHINESFILE"));
                pmake_machinesfile = GETNAME(wcs_buffer, FIND_LENGTH);
                append_makeflags_string(pmake_machinesfile, &makeflags_string);
                append_makeflags_string(pmake_machinesfile, &makeflags_string_po
        }
        /* -R */
        if (pmake_cap_r_specified) {
                append_char((int) space_char, &makeflags_string);
                append_char((int) hyphen_char, &makeflags_string);
                append_char('R', &makeflags_string);
                append_char((int) space_char, &makeflags_string_posix);
                append_char((int) hyphen_char, &makeflags_string_posix);
                append_char('R', &makeflags_string_posix);
        }

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

        if (makeflags_string.buffer.start[1] != (int) nul_char) {
                if (makeflags_string.buffer.start[1] != (int) space_char) {
                        MBSTOWCS(wcs_buffer, NOCATGETS("MFLAGS"));
                        (void) SETVAR(GETNAME(wcs_buffer, FIND_LENGTH),
                                      GETNAME(makeflags_string.buffer.start,
                                              FIND_LENGTH),
                                      false);
                } else {
                        MBSTOWCS(wcs_buffer, NOCATGETS("MFLAGS"));
                        (void) SETVAR(GETNAME(wcs_buffer, FIND_LENGTH),
                                      GETNAME(makeflags_string.buffer.start + 2,
                                              FIND_LENGTH),
                                      false);
                }
        }

/*
 *      Add command line macro to POSIX makeflags_string
 */
        if (makeflags_and_macro.start) {
                tmp_char = (char) space_char;
                cp = makeflags_and_macro.start;
                do {
                        append_char(tmp_char, &makeflags_string_posix);
                } while ( tmp_char = *cp++ );
                retmem_mb(makeflags_and_macro.start);
        }

/*
 *      Now set the value of MAKEFLAGS macro in accordance
 *      with current mode.
 */
        macro = maybe_append_prop(makeflags, macro_prop);
        temp = (Boolean) macro->body.macro.read_only;
        macro->body.macro.read_only = false;
        if(posix || gnu_style) {
                makeflags_string_current = &makeflags_string_posix;
        } else {
                makeflags_string_current = &makeflags_string;
        }
        if (makeflags_string_current->buffer.start[1] == (int) nul_char) {
                makeflags_value_saved =
                        GETNAME( makeflags_string_current->buffer.start + 1
                                , FIND_LENGTH
                                );
        } else {
                if (makeflags_string_current->buffer.start[1] != (int) space_cha
                        makeflags_value_saved =
                                GETNAME( makeflags_string_current->buffer.start
                                        , FIND_LENGTH
                                        );
                } else {
                        makeflags_value_saved =
                                GETNAME( makeflags_string_current->buffer.start
                                        , FIND_LENGTH
                                        );
```

```
                }
        }
        (void) SETVAR( makeflags
                     , makeflags_value_saved
                     , false
                     );
        macro->body.macro.read_only = temp;

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

        for (i = 1; i < argc; i++) {
                if (argv[i] &&
                    (argv[i][0] == (int) hyphen_char) &&
                    (argv[i][1] == 'f') &&
                    (argv[i][2] == (int) nul_char)) {
                        argv[i] = NULL;         /* Remove -f */
                        if (i >= argc - 1) {
                                fatal(catgets(catd, 1, 190, "No filename argumen
                        }
                        MBSTOWCS(wcs_buffer, argv[++i]);
                        primary_makefile = GETNAME(wcs_buffer, FIND_LENGTH);
                        (void) read_makefile(primary_makefile, true, true, true)
                        argv[i] = NULL;         /* Remove filename */
                        makefile_read = true;
                } else if (argv[i] &&
                           (argv[i][0] == (int) hyphen_char) &&
                           (argv[i][1] == 'c' ||
                            argv[i][1] == 'g' ||
                            argv[i][1] == 'j' ||
                            argv[i][1] == 'K' ||
                            argv[i][1] == 'M' ||
                            argv[i][1] == 'm' ||
                            argv[i][1] == 'O' ||
                            argv[i][1] == 'o') &&
                           (argv[i][2] == (int) nul_char)) {
                        argv[i] = NULL;
                        argv[++i] = NULL;
                }
        }

/*
 *      If no command line "-f" args then look for "makefile", and then for
 *      "Makefile" if "makefile" isn't found.
 */
        if (!makefile_read) {
                (void) read_dir(dot,
                                (wchar_t *) NULL,
                                (Property) NULL,
                                (wchar_t *) NULL);
            if (!posix) {
                if (makefile_name->stat.is_file) {
                        if (Makefile->stat.is_file) {
                                warning(catgets(catd, 1, 310, "Both `makefile' a
                        }
                        primary_makefile = makefile_name;
                        makefile_read = read_makefile(makefile_name,
                                                      false,
                                                      false,
                                                      true);
                }
                if (!makefile_read &&
                    Makefile->stat.is_file) {
                        primary_makefile = Makefile;
                        makefile_read = read_makefile(Makefile,
                                                      false,
                                                      false,
                                                      true);
                }
            } else {

                enum sccs_stat save_m_has_sccs = NO_SCCS;
                enum sccs_stat save_M_has_sccs = NO_SCCS;

                if (makefile_name->stat.is_file) {
                        if (Makefile->stat.is_file) {
                                warning(catgets(catd, 1, 191, "Both ‘makefile’ a
                        }
                }
                if (makefile_name->stat.is_file) {
                        if (makefile_name->stat.has_sccs == NO_SCCS) {
                           primary_makefile = makefile_name;
                           makefile_read = read_makefile(makefile_name,
                                                      false,
                                                      false,
                                                      true);
                        } else {
                          save_m_has_sccs = makefile_name->stat.has_sccs;
                          makefile_name->stat.has_sccs = NO_SCCS;
                          primary_makefile = makefile_name;
                          makefile_read = read_makefile(makefile_name,
                                                      false,
                                                      false,
                                                      true);
                        }
                }
                if (!makefile_read &&
                    Makefile->stat.is_file) {
                        if (Makefile->stat.has_sccs == NO_SCCS) {
                           primary_makefile = Makefile;
                           makefile_read = read_makefile(Makefile,
                                                      false,
                                                      false,
                                                      true);
                        } else {
                          save_M_has_sccs = Makefile->stat.has_sccs;
                          Makefile->stat.has_sccs = NO_SCCS;
                          primary_makefile = Makefile;
                          makefile_read = read_makefile(Makefile,
                                                      false,
                                                      false,
                                                      true);
                        }
                }
                if (!makefile_read &&
                        makefile_name->stat.is_file) {
                           makefile_name->stat.has_sccs = save_m_has_sccs;
                           primary_makefile = makefile_name;
                           makefile_read = read_makefile(makefile_name,
                                                      false,
                                                      false,
                                                      true);
                }
                if (!makefile_read &&
                    Makefile->stat.is_file) {
                           Makefile->stat.has_sccs = save_M_has_sccs;
```

```
                           primary_makefile = Makefile;
                           makefile_read = read_makefile(Makefile,
                                                      false,
                                                      false,
                                                      true);
                }
            }
        }
        do_not_exec_rule = save_do_not_exec_rule;
        allrules_read = makefile_read;
        trace_reader = false;

/*
 *      Now get current value of MAKEFLAGS and compare it with
 *      the saved value we set before reading makefile.
 *      If they are different then MAKEFLAGS is subsequently set by
 *      makefile, just leave it there. Otherwise, if make mode
 *      is changed by using .POSIX target in makefile we need
 *      to correct MAKEFLAGS value.
 */
        Name mf_val = getvar(makeflags);
        if( (posix != is_xpg4)
         && (!strcmp(mf_val->string_mb, makeflags_value_saved->string_mb)))
        {
                if (makeflags_string_posix.buffer.start[1] == (int) nul_char) {
                        (void) SETVAR(makeflags,
                                      GETNAME(makeflags_string_posix.buffer.star
                                              FIND_LENGTH),
                                      false);
                } else {
                        if (makeflags_string_posix.buffer.start[1] != (int) spac
                                (void) SETVAR(makeflags,
                                              GETNAME(makeflags_string_posix.buf
                                                      FIND_LENGTH),
                                              false);
                        } else {
                                (void) SETVAR(makeflags,
                                              GETNAME(makeflags_string_posix.buf
                                                      FIND_LENGTH),
                                              false);
                        }
                }
        }

        if (makeflags_string.free_after_use) {
                retmem(makeflags_string.buffer.start);
        }
        if (makeflags_string_posix.free_after_use) {
                retmem(makeflags_string_posix.buffer.start);
        }
        makeflags_string.buffer.start = NULL;
        makeflags_string_posix.buffer.start = NULL;

        if (posix) {
                /*
                 * If the user did not redefine the ARFLAGS macro in the
                 * default makefile (make.rules), then we’d like to
                 * change the macro value of ARFLAGS to be in accordance
                 * with "POSIX" requirements.
                 */
                MBSTOWCS(wcs_buffer, NOCATGETS("ARFLAGS"));
                name = GETNAME(wcs_buffer, wslen(wcs_buffer));
                macro = get_prop(name->prop, macro_prop);
                if ((macro != NULL) && /* Maybe (macro == NULL) || ? */
                    (IS_EQUAL(macro->body.macro.value->string_mb,
                              NOCATGETS("rv")))) {
```

```
                        MBSTOWCS(wcs_buffer, NOCATGETS("-rv"));
                        value = GETNAME(wcs_buffer, wslen(wcs_buffer));
                        (void) SETVAR(name,
                                      value,
                                      false);
                }
        }

        if (!posix && !svr4) {
                set_sgs_support();
        }


/*
 *      Make sure KEEP_STATE is in the environment if KEEP_STATE is on.
 */
        macro = get_prop(keep_state_name->prop, macro_prop);
        if ((macro != NULL) &&
            macro->body.macro.exported) {
                keep_state = true;
        }
        if (keep_state) {
                if (macro == NULL) {
                        macro = maybe_append_prop(keep_state_name,
                                                  macro_prop);
                }
                macro->body.macro.exported = true;
                (void) SETVAR(keep_state_name,
                              empty_name,
                              false);

                /*
                 *      Read state file
                 */

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

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

                if(!stat(make_state->string_mb, &make_state_stat)) {
                   if(!(make_state_stat.st_mode & S_IFREG) ) {
                        /* copy the make_state structure to the other
                        ** and then let make_state point to the new
                        ** one.
                        */
                      memcpy(&state_filename, make_state,sizeof(state_filename))
                      state_filename.string_mb = state_file_str_mb;
                /* Just a kludge to avoid two slashes back to back */
                      if((make_state->hash.length == 1)&&
                              (make_state->string_mb[0] == '/')) {
                         make_state->hash.length = 0;
                         make_state->string_mb[0] = '\0';
                      }
                      sprintf(state_file_str_mb,NOCATGETS("%s%s"),
                       make_state->string_mb,NOCATGETS("/.make.state"));
                      make_state = &state_filename;
                        /* adjust the length to reflect the appended string */
                      make_state->hash.length += 12;
                   }
```

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

                   if (slashp = strrchr(make_state->string_mb, '/')) {
                      strncpy(tmp_path, make_state->string_mb,
                                (slashp - make_state->string_mb));
                        tmp_path[slashp - make_state->string_mb]=0;
                      if(strlen(tmp_path)) {
                        if(stat(tmp_path, &make_state_stat)) {
                          warning(catgets(catd, 1, 192, "directory %s for .KEEP_
                        }
                        if (access(tmp_path, F_OK) != 0) {
                          warning(catgets(catd, 1, 193, "can't access dir %s"),t
                        }
                      }
                   }
                }
                if (report_dependencies_level != 1) {
                        Makefile_type   makefile_type_temp = makefile_type;
                        makefile_type = reading_statefile;
                        if (read_trace_level > 1) {
                                trace_reader = true;
                        }
                        (void) read_simple_file(make_state,
                                                false,
                                                false,
                                                false,
                                                false,
                                                false,
                                                true);
                        trace_reader = false;
                        makefile_type = makefile_type_temp;
                }
        }
}
```
______***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
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2005 Sun Microsystems, Inc. All rights reserved.
 * Use is subject to license terms.
 */


/*
 *      parallel.cc
 *
 *      Deal with the parallel processing
 */

/*
 * Included files
 */
#include <errno.h>              /* errno */
#include <fcntl.h>
#include <mk/defs.h>
#include <mksh/dosys.h>         /* redirect_io() */
#include <mksh/macro.h>         /* expand_value() */
#include <mksh/misc.h>          /* getmem() */
#include <sys/signal.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/utsname.h>
#include <sys/wait.h>
#include <unistd.h>
#include <netdb.h>



/*
 * Defined macros
 */
#define MAXRULES                100

/*
 * This const should be in avo_dms/include/AvoDmakeCommand.h
 */
const int local_host_mask = 0x20;
```

```
/*
 * typedefs & structs
 */


/*
 * Static variables
 */
static  Boolean         just_did_subtree = false;
static  char            local_host[MAXNAMELEN] = "";
static  char            user_name[MAXNAMELEN] = "";
static  int             pmake_max_jobs = 0;
static  pid_t           process_running = -1;
static  Running         *running_tail = &running_list;
static  Name            subtree_conflict;
static  Name            subtree_conflict2;


/*
 * File table of contents
 */
static  void            delete_running_struct(Running rp);
static  Boolean         dependency_conflict(Name target);
static  Doname          distribute_process(char **commands, Property line);
static  void            doname_subtree(Name target, Boolean do_get, Boolean impl
static  void            dump_out_file(char *filename, Boolean err);
static  void            finish_doname(Running rp);
static  void            maybe_reread_make_state(void);
static  void            process_next(void);
static  void            reset_conditionals(int cnt, Name *targets, Property *loc
static  pid_t           run_rule_commands(char *host, char **commands);
static  Property        *set_conditionals(int cnt, Name *targets);
static  void            store_conditionals(Running rp);


/*
 *      execute_parallel(line, waitflg)
 *
 *      DMake 2.x:
 *      parallel mode: spawns a parallel process to execute the command group.
 *      distributed mode: sends the command group down the pipe to rxm.
 *
 *      Return value:
 *                              The result of the execution
 *
 *      Parameters:
 *              line            The command group to execute
 */
Doname
execute_parallel(Property line, Boolean waitflg, Boolean local)
{
        int                     argcnt;
        int                     cmd_options = 0;
        char                    *commands[MAXRULES + 5];
        char                    *cp;
        Name                    dmake_name;
        Name                    dmake_value;
        int                     ignore;
        Name                    make_machines_name;
        char                    **p;
        Property                prop;
        Doname                  result = build_ok;
        Cmd_line                rule;
        Boolean                 silent_flag;
        Name                    target = line->body.line.target;
        Boolean                 wrote_state_file = false;

        if ((pmake_max_jobs == 0) &&
            (dmake_mode_type == parallel_mode)) {
                if (local_host[0] == '\0') {
                        (void) gethostname(local_host, MAXNAMELEN);
                }
                MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_MAX_JOBS"));
                dmake_name = GETNAME(wcs_buffer, FIND_LENGTH);
                if (((prop = get_prop(dmake_name->prop, macro_prop)) != NULL) &&
                    ((dmake_value = prop->body.macro.value) != NULL)) {
                        pmake_max_jobs = atoi(dmake_value->string_mb);
                        if (pmake_max_jobs <= 0) {
                                warning(catgets(catd, 1, 308, "DMAKE_MAX_JOBS ca
                                warning(catgets(catd, 1, 309, "setting DMAKE_MAX
                                pmake_max_jobs = PMAKE_DEF_MAX_JOBS;
                        }
                } else {
                        /*
                         * For backwards compatibility w/ PMake 1.x, when
                         * DMake 2.x is being run in parallel mode, DMake
                         * should parse the PMake startup file
                         * $(HOME)/.make.machines to get the pmake_max_jobs.
                         */
                        MBSTOWCS(wcs_buffer, NOCATGETS("PMAKE_MACHINESFILE"));
                        dmake_name = GETNAME(wcs_buffer, FIND_LENGTH);
                        if (((prop = get_prop(dmake_name->prop, macro_prop)) !=
                            ((dmake_value = prop->body.macro.value) != NULL)) {
                                make_machines_name = dmake_value;
                        } else {
                                make_machines_name = NULL;
                        }
                        if ((pmake_max_jobs = read_make_machines(make_machines_n
                                pmake_max_jobs = PMAKE_DEF_MAX_JOBS;
                        }
                }
        }

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

        {
                p = commands;
        }

        argcnt = 0;
        for (rule = line->body.line.command_used;
             rule != NULL;
             rule = rule->next) {
                if (posix && (touch || quest) && !rule->always_exec) {
                        continue;
                }
                if (vpath_defined) {
                        rule->command_line =
                          vpath_translation(rule->command_line);
                }

                if (dmake_mode_type == distributed_mode) {
                        cmd_options = 0;
                        if(local) {
                                cmd_options |= local_host_mask;
                        }
                } else {
                silent_flag = false;
```

```
                ignore = 0;

                }
                if (rule->command_line->hash.length > 0) {
                        if (++argcnt == MAXRULES) {
                                if (dmake_mode_type == distributed_mode) {
                                        /* XXX - tell rxm to execute on local ho
                                        /* I WAS HERE!!! */
                                } else {
                                        /* Too many rules, run serially instead.
                                return build_serial;
                        }
                        }
                        {
                                if (rule->silent && !silent) {
                                        silent_flag = true;
                                }
                                if (rule->ignore_error) {
                                        ignore++;
                                }
                                /* XXX - need to add support for + prefix */
                                if (silent_flag || ignore) {
                                        *p = getmem((silent_flag ? 1 : 0) +
                                                    ignore +
                                                    (strlen(rule->
                                                           command_line->
                                                           string_mb)) +
                                                    1);
                                        cp = *p++;
                                        if (silent_flag) {
                                                *cp++ = (int) at_char;
                                        }
                                        if (ignore) {
                                                *cp++ = (int) hyphen_char;
                                        }
                                        (void) strcpy(cp, rule->command_line->st
                                } else {
                                        *p++ = rule->command_line->string_mb;
                                }
                        }
                }
        }
        if ((argcnt == 0) ||
            (report_dependencies_level > 0)) {
                return build_ok;
        }
        {
                *p = NULL;

                Doname res = distribute_process(commands, line);
                if (res == build_running) {
                        parallel_process_cnt++;
                }

                /*
                 * Return only those memory that were specially allocated
                 * for part of commands.
                 */
                for (int i = 0; commands[i] != NULL; i++) {
                        if ((commands[i][0] == (int) at_char) ||
                            (commands[i][0] == (int) hyphen_char)) {
                                retmem_mb(commands[i]);
                        }
                }
                return res;
        }
```

```
}
______unchanged_portion_omitted_

/*
 *      finish_children(docheck)
 *
 *      Finishes the processing for all targets which were running
 *      and have now completed.
 *
 *      Parameters:
 *              docheck         Completely check the finished target
 *
 *      Static variables used:
 *              running_tail    The tail of the running list
 *
 *      Global variables used:
 *              continue_after_error  -k flag
 *              fatal_in_progress  True if we are finishing up after fatal err
 *              running_list    List of running processes
 */
void
finish_children(Boolean docheck)
{
        int             cmds_length;
        Property        line;
        Property        line2;
        struct stat     out_buf;
        Running         rp;
        Running         *rp_prev;
        Cmd_line        rule;
        Boolean         silent_flag;

        for (rp_prev = &running_list, rp = running_list;
             rp != NULL;
             rp = rp->next) {
bypass_for_loop_inc_4:
                /*
                 * If the state is ok or failed, then this target has
                 * finished building.
                 * In parallel_mode, output the accumulated stdout/stderr.
                 * Read the auto dependency stuff, handle a failed build,
                 * update the target, then finish the doname process for
                 * that target.
                 */
                if (rp->state == build_ok || rp->state == build_failed) {
                        *rp_prev = rp->next;
                        if (rp->next == NULL) {
                                running_tail = rp_prev;
                        }
                        if ((line2 = rp->command) == NULL) {
                                line2 = get_prop(rp->target->prop, line_prop);
                        }


                        if (dmake_mode_type == distributed_mode) {
                                if (rp->make_refd) {
                                        maybe_reread_make_state();
                                }
                        } else {
                        /*
                         * Check if there were any job output
                         * from the parallel build.
                         */
                        if (rp->stdout_file != NULL) {
                                if (stat(rp->stdout_file, &out_buf) < 0) {
                                        fatal(catgets(catd, 1, 130, "stat of %s
```

```
                                            rp->stdout_file,
                                            errmsg(errno));
                                }

#endif /* ! codereview */
                                if ((line2 != NULL) &&
                                    (out_buf.st_size > 0)) {
                                        cmds_length = 0;
                                        for (rule = line2->body.line.command_use
                                                 silent_flag = silent;
                                             rule != NULL;
                                             rule = rule->next) {
                                                cmds_length += rule->command_lin
                                                silent_flag = BOOLEAN(silent_fla
                                        }
                                        if (out_buf.st_size != cmds_length || si
                                            output_mode == txt2_mode) {
                                                dump_out_file(rp->stdout_file, f
                                        }
                                }
                                (void) unlink(rp->stdout_file);
                                retmem_mb(rp->stdout_file);
                                rp->stdout_file = NULL;
                        }

                        if (!out_err_same && (rp->stderr_file != NULL)) {
                                if (stat(rp->stderr_file, &out_buf) < 0) {
                                        fatal(catgets(catd, 1, 130, "stat of %s
                                            rp->stderr_file,
                                            errmsg(errno));
                                }
                                if ((line2 != NULL) &&
                                    (out_buf.st_size > 0)) {
                                        dump_out_file(rp->stderr_file, true);
                                }
                                (void) unlink(rp->stderr_file);
                                retmem_mb(rp->stderr_file);
                                rp->stderr_file = NULL;
                        }

                        }
                        check_state(rp->temp_file);
                        if (rp->temp_file != NULL) {
                                free_name(rp->temp_file);
                        }
                        rp->temp_file = NULL;
                        if (rp->state == build_failed) {
                                line = get_prop(rp->target->prop, line_prop);
                                if (line != NULL) {
                                        line->body.line.command_used = NULL;
                                }
                                if (continue_after_error ||
                                    fatal_in_progress ||
                                    !docheck) {
                                        warning(catgets(catd, 1, 256, "Command f
                                                rp->command ? line2->body.line.t
                                        build_failed_seen = true;
                                } else {
                                        /*
                                         * XXX??? - DMake needs to exit(),
                                         * but shouldn't call fatal().
                                         */
#ifdef PRINT_EXIT_STATUS
                                        warning(NOCATGETS("I'm in finish_childre
#endif
```

```
                                        fatal(catgets(catd, 1, 258, "Command fai
                                                rp->command ? line2->body.line.t
                                }
                        }
                        if (!docheck) {
                                delete_running_struct(rp);
                                rp = *rp_prev;
                                if (rp == NULL) {
                                        break;
                                } else {
                                        goto bypass_for_loop_inc_4;
                                }
                        }
                        update_target(get_prop(rp->target->prop, line_prop),
                                      rp->state);
                        finish_doname(rp);
                        delete_running_struct(rp);
                        rp = *rp_prev;
                        if (rp == NULL) {
                                break;
                        } else {
                                goto bypass_for_loop_inc_4;
                        }
                } else {
                        rp_prev = &rp->next;
                }
        }
}
```
______***unchanged_portion_omitted_***

****
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_***

```
typedef enum {
        serial_mode,
        parallel_mode
        parallel_mode,
        distributed_mode
} DMake_mode;
```
______***unchanged_portion_omitted_***