

/* Miscellaneous functions, not really specific to GNU tar.

Copyright (C) 1988, 1992, 1994, 1995, 1996, 1997, 1999, 2000, 2001,
2003, 2004, 2005, 2006, 2007 Free Software Foundation, Inc.

This program is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 3, or (at your option) any later
version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA. */

```
#include <system.h>
#include <rmt.h>
#include "common.h"
#include <quotearg.h>
#include <save-cwd.h>
#include <xgetcwd.h>
#include <unlinkdir.h>
#include <utimens.h>
```

```
#if HAVE_STROPTS_H
# include <stropts.h>
#endif
#if HAVE_SYS_FILIO_H
# include <sys/filio.h>
#endif
```

```
/* Handling strings. */
```

```
/* Assign STRING to a copy of VALUE if not zero, or to zero. If  
STRING was nonzero, it is freed first. */
```

```
void  
assign_string (char **string, const char *value)  
{  
    if (*string)  
        free (*string);  
    *string = value ? xstrdup (value) : 0;  
}
```

```
/* Allocate a copy of the string quoted as in C, and returns that. If  
the string does not have to be quoted, it returns a null pointer.  
The allocated copy should normally be freed with free() after the  
caller is done with it.
```

This is used in one context only: generating the directory file in incremental dumps. The quoted string is not intended for human consumption; it is intended only for `unquote_string`. The quoting is locale-independent, so that users needn't worry about locale when reading directory files. This means that we can't use `quotearg`, as `quotearg` is locale-dependent and is meant for human consumption. */

```
char *  
quote_copy_string (const char *string)  
{  
    const char *source = string;  
    char *destination = 0;  
    char *buffer = 0;  
    int copying = 0;  
  
    while (*source)  
    {  
        int character = *source++;  
  
        switch (character)  
        {  
            case '\n': case '\\':  
                if (!copying)  
                {  
                    size_t length = (source - string) - 1;  
  
                    copying = 1;  
                    buffer = xmalloc (length + 2 + 2 * strlen (source) + 1);  
                    memcpy (buffer, string, length);  
                    destination = buffer + length;  
                }  
                *destination++ = '\\';  
                *destination++ = character == '\\'? '\\': 'n';  
                break;
```

```

        default:
            if (copying)
                *destination++ = character;
            break;
        }
    }
}
if (copying)
{
    *destination = '\0';
    return buffer;
}
return 0;
}

```

/* Takes a quoted C string (like those produced by quote_copy_string) and turns it back into the un-quoted original. This is done in place. Returns 0 only if the string was not properly quoted, but completes the unquoting anyway.

This is used for reading the saved directory file in incremental dumps. It is used for decoding old `N' records (demangling names). But also, it is used for decoding file arguments, would they come from the shell or a -T file, and for decoding the --exclude argument. */

```

int
unquote_string (char *string)
{
    int result = 1;
    char *source = string;
    char *destination = string;

```

/* Escape sequences other than \\ and \n are no longer generated by quote_copy_string, but accept them for backwards compatibility, and also because unquote_string is used for purposes other than parsing the output of quote_copy_string. */

```

while (*source)
    if (*source == '\\')
        switch (*++source)
        {
            case '\\':
                *destination++ = '\\';
                source++;
                break;

            case 'a':
                *destination++ = '\a';
                source++;
                break;

            case 'b':

```

```
*destination++ = '\b';
source++;
break;

case 'f':
    *destination++ = '\f';
    source++;
    break;

case 'n':
    *destination++ = '\n';
    source++;
    break;

case 'r':
    *destination++ = '\r';
    source++;
    break;

case 't':
    *destination++ = '\t';
    source++;
    break;

case 'v':
    *destination++ = '\v';
    source++;
    break;

case '?':
    *destination++ = 0177;
    source++;
    break;

case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
    {
        int value = *source++ - '0';

        if (*source < '0' || *source > '7')
            {
                *destination++ = value;
                break;
            }
        value = value * 8 + *source++ - '0';
        if (*source < '0' || *source > '7')
```

```
    {
        *destination++ = value;
        break;
    }
    value = value * 8 + *source++ - '0';
    *destination++ = value;
    break;
}

default:
    result = 0;
    *destination++ = '\\';
    if (*source)
        *destination++ = *source++;
    break;
}
else if (source != destination)
    *destination++ = *source++;
else
    source++, destination++;

if (source != destination)
    *destination = '\0';
return result;
}
```

```

/* Handling numbers. */

/* Output fraction and trailing digits appropriate for a nanoseconds
count equal to NS, but don't output unnecessary '.' or trailing
zeros. */

void
code_ns_fraction (int ns, char *p)
{
    if (ns == 0)
        *p = '\0';
    else
        {
            int i = 9;
            *p++ = '.';

            while (ns % 10 == 0)
                {
                    ns /= 10;
                    i--;
                }

            p[i] = '\0';

            for (;;)
                {
                    p[--i] = '0' + ns % 10;
                    if (i == 0)
                        break;
                    ns /= 10;
                }
        }
}

char const *
code_timespec (struct timespec t, char sbuf[TIMESPEC_STRSIZE_BOUND])
{
    time_t s = t.tv_sec;
    int ns = t.tv_nsec;
    char *np;
    bool negative = s < 0;

    if (negative && ns != 0)
        {
            s++;
            ns = BILLION - ns;
        }

    np = umaxtostr (negative ? - (uintmax_t) s : (uintmax_t) s, sbuf + 1);
    if (negative)
        *--np = '-';
}

```

```
code_ns_fraction (ns, sbuf + UINTMAX_STRSIZE_BOUND);  
return np;  
}
```

```

/* File handling. */

/* Saved names in case backup needs to be undone. */
static char *before_backup_name;
static char *after_backup_name;

/* Return 1 if FILE_NAME is obviously "." or "/". */
static bool
must_be_dot_or_slash (char const *file_name)
{
  file_name += FILE_SYSTEM_PREFIX_LEN (file_name);

  if (ISSLASH (file_name[0]))
    {
      for (;;)
        if (ISSLASH (file_name[1]))
          file_name++;
        else if (file_name[1] == '.'
                 && ISSLASH (file_name[2 + (file_name[2] == '.')]))
          file_name += 2 + (file_name[2] == '.');
        else
          return ! file_name[1];
    }
  else
    {
      while (file_name[0] == '.' && ISSLASH (file_name[1]))
        {
          file_name += 2;
          while (ISSLASH (*file_name))
            file_name++;
        }

      return ! file_name[0] || (file_name[0] == '.' && ! file_name[1]);
    }
}

/* Some implementations of rmdir let you remove '.' or '/'.
   Report an error with errno set to zero for obvious cases of this;
   otherwise call rmdir. */
static int
safer_rmdir (const char *file_name)
{
  if (must_be_dot_or_slash (file_name))
    {
      errno = 0;
      return -1;
    }

  return rmdir (file_name);
}

```

```

/* Remove FILE_NAME, returning 1 on success. If FILE_NAME is a directory,
then if OPTION is RECURSIVE_REMOVE_OPTION is set remove FILE_NAME
recursively; otherwise, remove it only if it is empty. If FILE_NAME is
a directory that cannot be removed (e.g., because it is nonempty)
and if OPTION is WANT_DIRECTORY_REMOVE_OPTION, then return -1.
Return 0 on error, with errno set; if FILE_NAME is obviously the working
directory return zero with errno set to zero. */
int
remove_any_file (const char *file_name, enum remove_option option)
{
/* Try unlink first if we cannot unlink directories, as this saves
us a system call in the common case where we're removing a
non-directory. */
bool try_unlink_first = cannot_unlink_dir ();

if (try_unlink_first)
{
if (unlink (file_name) == 0)
return 1;

/* POSIX 1003.1-2001 requires EPERM when attempting to unlink a
directory without appropriate privileges, but many Linux
kernels return the more-sensible EISDIR. */
if (errno != EPERM && errno != EISDIR)
return 0;
}

if (safer_rmdir (file_name) == 0)
return 1;

switch (errno)
{
case ENOTDIR:
return !try_unlink_first && unlink (file_name) == 0;

case 0:
case EEXIST:
#if defined ENOTEMPTY && ENOTEMPTY != EEXIST
case ENOTEMPTY:
#endif
switch (option)
{
case ORDINARY_REMOVE_OPTION:
break;

case WANT_DIRECTORY_REMOVE_OPTION:
return -1;

case RECURSIVE_REMOVE_OPTION:
{
char *directory = savedir (file_name);
char const *entry;

```

```

size_t entrylen;

if (! directory)
    return 0;

for (entry = directory;
     (entrylen = strlen (entry)) != 0;
     entry += entrylen + 1)
{
    char *file_name_buffer = new_name (file_name, entry);
    int r = remove_any_file (file_name_buffer,
                            RECURSIVE_REMOVE_OPTION);
    int e = errno;
    free (file_name_buffer);

    if (! r)
    {
        free (directory);
        errno = e;
        return 0;
    }
}

free (directory);
return safer_rmdir (file_name) == 0;
}
}
break;
}

return 0;
}

/* Check if FILE_NAME already exists and make a backup of it right now.
Return success (nonzero) only if the backup is either unneeded, or
successful. For now, directories are considered to never need
backup. If THIS_IS_THE_ARCHIVE is nonzero, this is the archive and
so, we do not have to backup block or character devices, nor remote
entities. */
bool
maybe_backup_file (const char *file_name, bool this_is_the_archive)
{
    struct stat file_stat;

    /* Check if we really need to backup the file. */

    if (this_is_the_archive && _remdev (file_name))
        return true;

    if (stat (file_name, &file_stat))
    {
        if (errno == ENOENT)

```

```

    return true;

    stat_error (file_name);
    return false;
}

if (S_ISDIR (file_stat.st_mode))
    return true;

if (this_is_the_archive
    && (S_ISBLK (file_stat.st_mode) || S_ISCHR (file_stat.st_mode)))
    return true;

assign_string (&before_backup_name, file_name);

/* A run situation may exist between Emacs or other GNU programs trying to
   make a backup for the same file simultaneously. If theoretically
   possible, real problems are unlikely. Doing any better would require a
   convention, GNU-wide, for all programs doing backups. */

assign_string (&after_backup_name, 0);
after_backup_name = find_backup_file_name (file_name, backup_type);
if (! after_backup_name)
    xalloc_die ();

if (rename (before_backup_name, after_backup_name) == 0)
{
    if (verbose_option)
        fprintf (stderr, _("Renaming %s to %s\n"),
                 quote_n (0, before_backup_name),
                 quote_n (1, after_backup_name));
    return true;
}
else
{
    /* The backup operation failed. */
    int e = errno;
    ERROR ((0, e, _("%s: Cannot rename to %s"),
           quotearg_colon (before_backup_name),
           quote_n (1, after_backup_name)));
    assign_string (&after_backup_name, 0);
    return false;
}
}

/* Try to restore the recently backed up file to its original name.
   This is usually only needed after a failed extraction. */
void
undo_last_backup (void)
{
    if (after_backup_name)
    {

```

```

if (rename (after_backup_name, before_backup_name) != 0)
    {
        int e = errno;
        ERROR ((0, e, _("%s: Cannot rename to %s"),
            quotearg_colon (after_backup_name),
            quote_n (1, before_backup_name)));
    }
if (verbose_option)
    fprintf (stdlis, _("Renaming %s back to %s\n"),
        quote_n (0, after_backup_name),
        quote_n (1, before_backup_name));
assign_string (&after_backup_name, 0);
}
}

/* Depending on DEREf, apply either stat or lstat to (NAME, BUF). */
int
deref_stat (bool deref, char const *name, struct stat *buf)
{
    return deref ? stat (name, buf) : lstat (name, buf);
}

/* Set FD's (i.e., FILE's) access time to TIMESPEC[0]. If that's not
   possible to do by itself, set its access and data modification
   times to TIMESPEC[0] and TIMESPEC[1], respectively. */
int
set_file_atime (int fd, char const *file, struct timespec const timespec[2])
{
#ifdef _FIOSTATIME
    if (0 <= fd)
        {
            struct timeval timeval;
            timeval.tv_sec = timespec[0].tv_sec;
            timeval.tv_usec = timespec[0].tv_nsec / 1000;
            if (ioctl (fd, _FIOSTATIME, &timeval) == 0)
                return 0;
        }
#endif
return gl_futimens (fd, file, timespec);
}

/* A description of a working directory. */
struct wd
{
    char const *name;
    int saved;
    struct saved_cwd saved_cwd;
};

/* A vector of chdir targets. wd[0] is the initial working directory. */
static struct wd *wd;

```

```

/* The number of working directories in the vector. */
static size_t wds;

/* The allocated size of the vector. */
static size_t wd_alloc;

/* DIR is the operand of a -C option; add it to vector of chdir targets,
   and return the index of its location. */
int
chdir_arg (char const *dir)
{
  if (wds == wd_alloc)
    {
      if (wd_alloc == 0)
        {
          wd_alloc = 2;
          wd = xmalloc (sizeof *wd * wd_alloc);
        }
      else
        wd = x2nrealloc (wd, &wd_alloc, sizeof *wd);

      if (! wds)
        {
          wd[wds].name = ".";
          wd[wds].saved = 0;
          wds++;
        }
    }

  /* Optimize the common special case of the working directory,
     or the working directory as a prefix. */
  if (dir[0])
    {
      while (dir[0] == '.' && ISSLASH (dir[1]))
        for (dir += 2; ISSLASH (*dir); dir++)
          continue;
      if (! dir[dir[0] == '.'])
        return wds - 1;
    }

  wd[wds].name = dir;
  wd[wds].saved = 0;
  return wds++;
}

/* Change to directory I. If I is 0, change to the initial working
   directory; otherwise, I must be a value returned by chdir_arg. */
void
chdir_do (int i)
{
  static int previous;

```

```

if (previous != i)
{
    struct wd *prev = &wd[previous];
    struct wd *curr = &wd[i];

    if (! prev->saved)
    {
        int err = 0;
        prev->saved = 1;
        if (save_cwd (&prev->saved_cwd) != 0)
            err = errno;
        else if (0 <= prev->saved_cwd.desc)
        {
            /* Make sure we still have at least one descriptor available. */
            int fd1 = prev->saved_cwd.desc;
            int fd2 = dup (fd1);
            if (0 <= fd2)
                close (fd2);
            else if (errno == EMFILE)
            {
                /* Force restore_cwd to use chdir_long. */
                close (fd1);
                prev->saved_cwd.desc = -1;
                prev->saved_cwd.name = xgetcwd ();
            }
            else
                err = errno;
        }

        if (err)
            FATAL_ERROR ((0, err, _("Cannot save working directory")));
    }

    if (curr->saved)
    {
        if (restore_cwd (&curr->saved_cwd))
            FATAL_ERROR ((0, 0, _("Cannot change working directory")));
    }
    else
    {
        if (i && ! ISSLASH (curr->name[0]))
            chdir_do (i - 1);
        if (chdir (curr->name) != 0)
            chdir_fatal (curr->name);
    }

    previous = i;
}
}

```

```
void
close_diag (char const *name)
{
    if (ignore_failed_read_option)
        close_warn (name);
    else
        close_error (name);
}
```

```
void
open_diag (char const *name)
{
    if (ignore_failed_read_option)
        open_warn (name);
    else
        open_error (name);
}
```

```
void
read_diag_details (char const *name, off_t offset, size_t size)
{
    if (ignore_failed_read_option)
        read_warn_details (name, offset, size);
    else
        read_error_details (name, offset, size);
}
```

```
void
readlink_diag (char const *name)
{
    if (ignore_failed_read_option)
        readlink_warn (name);
    else
        readlink_error (name);
}
```

```
void
savedir_diag (char const *name)
{
    if (ignore_failed_read_option)
        savedir_warn (name);
    else
        savedir_error (name);
}
```

```
void
seek_diag_details (char const *name, off_t offset)
{
    if (ignore_failed_read_option)
        seek_warn_details (name, offset);
    else
```

```
    seek_error_details (name, offset);
}
```

```
void
stat_diag (char const *name)
{
    if (ignore_failed_read_option)
        stat_warn (name);
    else
        stat_error (name);
}
```

```
void
write_fatal_details (char const *name, ssize_t status, size_t size)
{
    write_error_details (name, status, size);
    fatal_exit ();
}
```

```
/* Fork, aborting if unsuccessful. */
pid_t
xfork (void)
{
    pid_t p = fork ();
    if (p == (pid_t) -1)
        call_arg_fatal ("fork", _("child process"));
    return p;
}
```

```
/* Create a pipe, aborting if unsuccessful. */
void
xpipe (int fd[2])
{
    if (pipe (fd) < 0)
        call_arg_fatal ("pipe", _("interprocess channel"));
}
```

```
/* Return PTR, aligned upward to the next multiple of ALIGNMENT.
   ALIGNMENT must be nonzero. The caller must arrange for ((char *)
   PTR) through ((char *) PTR + ALIGNMENT - 1) to be addressable
   locations. */
```

```
static inline void *
ptr_align (void *ptr, size_t alignment)
{
    char *p0 = ptr;
    char *p1 = p0 + alignment - 1;
    return p1 - (size_t) p1 % alignment;
}
```

```
/* Return the address of a page-aligned buffer of at least SIZE bytes.
   The caller should free *PTR when done with the buffer. */
```

```
void *
page_aligned_alloc (void **ptr, size_t size)
{
    size_t alignment = getpagesize ();
    size_t size1 = size + alignment;
    if (size1 < size)
        xalloc_die ();
    *ptr = xmalloc (size1);
    return ptr_align (*ptr, alignment);
}
```