File System Kernel and Library Calls

COMP 321

Dave Johnson

 Copyright © 2026 David B. Johnson

Creating a New Hard Link

```
int link(const char *oldpath, const char *newpath);
```

Creates a new hard link to the existing file indicated by oldpath

- newpath is the name for the new hard link
  - The directory for newpath must already exist
  - link( ) creates only a single new directory entry – and it has the same inode number in it as the existing inode for oldpath
- Example: link("/abc/def", "123/4567/890")
  - The directory "123/4567" must already exist
  - Creates the new name "890" in that directory
- If newpath already exists, returns -1 with errno = EEXIST

 Copyright © 2026 David B. Johnson

Creating a New Symbolic Link

```
int symlink(const char *target, const char *linkpath);
```

Creates a new symbolic link to the given target name

- linkpath is the name for the new symbolic link
  - The directory for linkpath must already exist
  - symlink( ) creates only a single new directory entry
- Example: symlink("abc/def", "123/4567/890")
  - The directory "123/4567" must already exist
  - Creates the new name "890" in that directory
- The file target normally should exist but does not need to exist now
- If linkpath already exists, returns -1 with errno = EEXIST

Copyright © 2026 David B. Johnson

Removing a Hard Link (and a File)

```
int unlink(const char *pathname);
```

Removes the hard link given as the last component of pathname

- Removes only that single directory entry from the given directory
- Example: unlink("123/4567/890")
  - Removes the name "890" in that directory
- If pathname refers to a directory, returns -1 with errno = EISDIR
- If pathname refers to a symbolic link, the symbolic link (not the target of that link) is removed
- If this is the last hard link to that inode, the inode and data blocks are freed
  - But if any file descriptors are open to it, ***not until they are all closed***

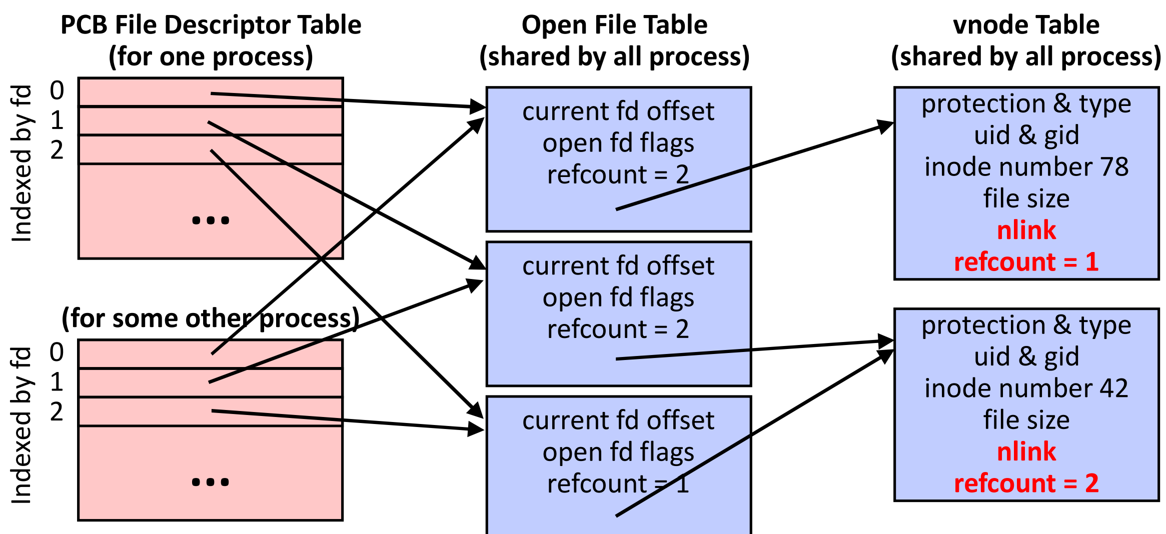
Copyright © 2026 David B. Johnson

Unlinking a File While Open

Commonly used for making a "temporary" file

- If one or more file descriptors (in any process) are open to some file – actually, that means open to that inode
- And the last directory entry with that inode number in it is removed due to some call to unlink
- The inode and data blocks are ***not*** freed then
- But they ***will*** be freed when the last file descriptor is closed
  - The file descriptors still work exactly as normal until then
- Can be used to make a temporary file that goes away ***automatically*** when no longer in use – even if the process crashes or forgets to delete the file later
  - Process calls create() and then calls unlink() right away

Review: Kernel File Data Structures



Unlinking a File While Open

The kernel file data structures track all the information needed

- A copy of the disk inode is kept in the vnode table entry for that file
  - And the inode nlink field is the count of (remaining) hard links to it
- The vnode table entry also contains a reference count
  - A count of open file table entries pointing to that vnode table entry
  - Already needed so the kernel knows when to free the vnode table entry from memory
- The open file table entry contains a reference count
  - A count of file descriptors pointing to that open file table entry
- When the vnode table entry in memory reference count goes to 0
  - If the inode nlink is also 0, then the inode and data blocks are freed

Reading a Symbolic Link's Target Name

```
ssize_t  readlink(const char  *restrict pathname,
         char  *restrict buf,  size_t  bufsiz);
```

Returns the target name for pathname into the buffer buf

- Roughly like fd = open(pathname), then read( ) and close( ) that fd
  - But doing that would instead open and read from the target file itself
  - readlink( ) does not "follow through" the symbolic link, but instead accesses the symbolic link itself
- readlink( ) returns the length of the target name
  - Does not add a null '\0' byte to the end in buf
  - And if bufsiz isn't large enough, readlink() silently truncates the part of the target name returned in buf

Renaming a File

```
int rename(const char *oldpath, const char *newpath);
```

Changes the name of an existing file and/or moves the file

- The directories leading to the last component in oldpath and newpath must all already exist
  - Only modifies the last component in the oldpath and newpath directories
- May change the name within the same directory
  - Example: rename("abc/def/ghi", "abc/def/xyz")
- And/or may change that name into a different directory
  - Example: rename("abc/def/ghi", "/123/456/ghi")
- If oldpath or newpath refer to a symbolic link, it operates on the symbolic link

Copyright © 2026 David B. Johnson

Making a Directory

```
int mkdir(const char *pathname, mode_t mode);
```

Makes a new directory given by pathname

- The directories leading to last component in pathname must all already exist
  - Only makes the new directory itself there
- Automatically creates the "." and ".." links in that new directory
- The protection for the new directory is given by "mode"
- Example: mkdir("/abc/def/ghi")
  - Makes the new directory "ghi" in the existing directory "/abc/def"

Copyright © 2026 David B. Johnson

Removing a Directory

```
int  rmdir(const char  *pathname);
```

Removes the existing directory given by pathname

- Only removes the last component of pathname, which must be a directory
- Example: rmdir("/abc/def/ghi")
  - Removes the directory "ghi" from the directory "/abc/def"
- The directory to be removed must be "empty"
  - It must contain ***only*** that directory's "." and ".." entries
  - (and possibly "deleted entries with the inode number set to 0)
  - The nlink for the directory ***will be*** 2 (name from above and its "." entry)
- Frees the directory's inode and its data blocks

Setting a Process's Current Directory

```
int  chdir(const char  *path);
```

Changes the calling process's current directory to "path"

- That directory becomes its starting point for looking up relative pathnames
- The current directory of a process is remembered in its PCB as the ***inode number*** and ***device id*** of that directory (***not*** the directory's name, e.g., path)
  - That directory is found and remembered at chdir( ) time
  - It thus does not matter if later, any directory is renamed
  - It thus does not matter if later, permissions are changed to no longer allow the process access to the directories in "path"
- Called by the shell built-in command "cd"

Reading a Directory (Like What “ls” Does)

The OLD way was easy: just open( ) it and read( ) it

- The contents bytes were in the form of directory entries, such as

```
struct   direct                          /* total of 16 bytes per entry */
{
         ino_t    d_ino;                 /* inode number of the file */
         char     d_name[14];            /* the (or, a) name for the file */
};
```

- Simple, but it means the format of a directory (and of all directory entries) must be known by “everybody”
  - Very hard to evolve the format (e.g., newer versions of Unix/Linux), such as including longer filenames or extra information in a directory entry
- ***Now***, any attempt to read( ) from a directory returns -1 with errno = EISDIR

Copyright © 2026 David B. Johnson

Reading a Directory (Like What “ls” Does)

The NEW way uses a library interface similar to stdio

- An open directory stream is represented by a “DIR *” (like stdio “FILE *”)
- Reading from the “DIR *” stream returns an “***imaginary***” ***(abstract)*** representation for each directory entry – this is the Linux definition

```
struct dirent {
    ino_t      d_ino;             /* inode number */
    off_t      d_off;             /* identification of this “location” in the directory */
    unsigned short  d_reclen;     /* length of this record */
    unsigned char   d_type;       /* type of file (for supported filesystems) */
    char       d_name[256];       /* null-terminated filename */
};
```

- This is returned to the library from a kernel call that should ***not*** otherwise be used (see “man ***3*** readdir”, ***not*** “man ***2*** readdir” and ***not*** “man ***2*** getdents”)

Copyright © 2026 David B. Johnson

Opening a Directory

```
DIR  *opendir(const char  *name);
DIR  *fdopendir(int  fd);
```

Two ways to get a "DIR *" for operations on a directory

- Open it by pathname "name"
  - The "DIR *" stream starts positioned at the first directory entry
- Open it by, e.g., open( ) to get a file descriptor and then access by a "DIR *"
  - Yes, you can still open( ) a directory, even though can't read( ) from it
  - The "DIR *" stream starts at a position determined by the current position for file descriptor fd
- On error, NULL is returned and errno is set

Copyright © 2026 David B. Johnson

Reading from a Directory

```
struct dirent  *readdir(DIR  *dirp);
```

Each readdir( ) call returns the next single directory entry in this directory

- Returns a pointer to an image of that directory entry
  - It may be statically allocated, do not try to free( ) it
  - This memory may be overwritten by the next call to readdir( ) for the same directory stream
- Returns NULL if there are no more entries in this directory
  - ***Also*** returns NULL on any error, with errno set to show which error
  - Recommendation: set errno = 0 before readdir( ) call to tell the difference
- Again, see "man ***3*** readdir", ***not*** "man ***2*** readdir" and ***not*** "man ***2*** getdents"

Copyright © 2026 David B. Johnson

The d_reclen Field Returned by readdir( )

The format retuned by readdir( ) contains one (whole) directory entry

- Again, the format (in Linux) returned looks like this

```
struct dirent {
    ino_t      d_ino;              /* inode number */
    off_t      d_off;              /* identification of this "location" in the directory */
    unsigned short  d_reclen;      /* length of this record */
    unsigned char   d_type;        /* type of file (for supported filesystems) */
    char       d_name[256];        /* null-terminated filename */
};
```

- The size of the d_name field may not really be 256, but it holds the whole name (and a '\0' null termination)
- The d_reclen field is the big enough for this entire (abstract) struct dirent

Copyright © 2026 David B. Johnson

The d_type Field Returned by readdir( )

Similar to the S_IFMT part of st_mode returned by stat( ) and fstat( )

- DT_BLK This is a block device
- DT_CHR This is a character device
- DT_DIR This is a directory
- DT_FIFO This is a named pipe (FIFO)
- DT_LNK This is a symbolic link
- DT_REG This is a regular file
- DT_SOCK This is a UNIX domain socket
- DT_UNKNOWN The file type could not be determined

Available for most filesystem types (but may be DT_UNKNOWN for others)

Copyright © 2026 David B. Johnson

Moving Around within a “DIR *” Directory Stream

```
long telldir(DIR *dirp);
void seekdir(DIR *dirp, long loc);
void rewinddir(DIR *dirp);
```

- telldir( ) returns an abstract indication of the current location in the stream
  - The d_off field returned by readdir( ) is the same kind of abstract location
  - Both are usable only for future calls to seekdir( ) on this “DIR *” stream
  - These are ***not*** (necessarily) a byte offset within the data in the directory
- seekdir( ) moves the “DIR *” stream to the indicated abstract location within it
- rewinddir( ) moves the “DIR *” stream to the beginning (first directory entry)

Copyright © 2026 David B. Johnson

Closing an Open Directory Stream

```
int closedir(DIR *dirp);
```

Closes the directory stream and frees any resources from the open

- Frees the memory allocated at open at the “DIR *” address
- And closes the corresponding file descriptor
  - From an opendir( ), that file descriptor was internally opened
  - From an fdopendir( ), that file descriptor was already open then but is still closed on the closedir( ) call
- The “DIR *” pointer dirp is not useable after this call
- Returns -1 on any error, with errno set appropriately

Copyright © 2026 David B. Johnson