

Unix File System API

**Operating System
Hebrew University
Spring 2009**

File System API manuals

- The information on file system API can be found on section 2 (Unix and C system calls) of the **Unix/Linux man pages**

```
~> man -S 2 open
```

```
~> man -S 2 read
```

And so on...

open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname,
         int oflag,
         /* mode_t mode */);
```

open - oflag

- O_RDONLY open for reading only
- O_WRONLY open for writing only
- O_RDWR open for reading and writing
- O_APPEND append on each write –
not atomic when using NFS
- O_CREAT create file if it does not exist
- O_TRUNC truncate size to 0
- O_EXCL error if create and file exists
- O_SYNC Any **writes** on the resulting file descriptor will block the calling process until the data has been **physically written** to the underlying hardware .

open - mode

- Specifies the permissions to use if a new file is created.
- This mode only applies to **future accesses** of the newly created file.

User: S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR

Group: S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP

Other: S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH

- mode must be specified when O_CREAT is in the flags.

Identifying errors

- How can we tell if the call failed?
 - the system call returns a negative number
- How can we tell **what** was the error?
 - Using **errno** – a global variable set by the system call if an error has occurred.
 - Defined in **errno.h**
 - Use **strerror** to get a string describing the problem
 - Use **perror** to print a string describing the problem

```
#include <errno.h>
int fd;
fd = open( FILE_NAME, O_RDONLY, 0644 );
if( fd < 0 ) {
    printf( "Error opening file: %s\n", strerror( errno ) );
    return -1;
}
```

open – possible errno values

- EEXIST – O_CREAT and O_EXCL were specified and the file exists.
- ENAMETOOLONG - A component of a pathname exceeded {NAME_MAX} characters, or an entire path name exceeded {PATH_MAX} characters.
- ENOENT - O_CREAT is not set and the named file does not exist.
- ENOTDIR - A component of the path prefix is not a directory.
- EROFS - The named file resides on a read-only file system, and write access was requested.
- ENOSPC - O_CREAT is specified, the file does not exist, and there is no space left on the file system containing the directory.
- EMFILE - The process has already reached its limit for open file descriptors.

creat

```
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>
```

```
int creat(const char *pathname, mode_t mode)
```

Equivalent to:

```
open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)
```

close

```
#include <unistd.h>
```

```
int close(int fd)
```

lseek

```
#include <sys/types.h>
#include <unistd.h>
```

```
off_t lseek(int fd, off_t offset, int whence);
```

- **fd**
 - The file descriptor.
 - It must be an open file descriptor.
- **offset**
 - Repositions the offset of the file descriptor **fd** to the argument **offset** according to the directive **whence**.
- **Return value**
 - The offset in the file after the seek
 - If negative, **errno** is set.

lseek – whence

- `SEEK_SET` - the offset is set to *offset* bytes from the beginning of the file.
- `SEEK_CUR` - the offset is set to its current location plus *offset* bytes.
 - `currpos = lseek(fd, 0, SEEK_CUR)`
- `SEEK_END` - the offset is set to the size of the file plus *offset* bytes.
 - If we use `SEEK_END` and then write to the file, it extends the file size in kernel and the gap is filled with zeros.

lseek: Examples

- Move to byte #16
 - `newpos = lseek(fd, 16, SEEK_SET);`
- Move forward 4 bytes
 - `newpos = lseek(fd, 4, SEEK_CUR);`
- Move to 8 bytes from the end
 - `newpos = lseek(fd, -8, SEEK_END);`
- Move backward 3 bytes
 - `lseek(fd, -3, SEEK_CUR);`

lseek - errno

- **lseek()** will fail and the file pointer will remain unchanged if:
 - EBADF - *fd* is not an open file descriptor.
 - ESPIPE - *fd* is associated with a pipe, socket, or FIFO.
 - EINVAL - *Whence* is not a proper value.

read

```
#include <unistd.h>
```

```
ssize_t read(int fd, void *buff, size_t nbytes)
```

- Attempts to read *nbytes* of data from the object referenced by the descriptor *fd* into the buffer pointed to by *buff*.
- If successful, the number of bytes **actually read** is returned.
- If we are at end-of-file, zero is returned.
- Otherwise, -1 is returned and the global variable *errno* is set to indicate the error.

read - errno

- EBADF - *fd* is not a valid file descriptor or it is not open for reading.
- EIO - An I/O error occurred while reading from the file system.
- EINTR The call was interrupted by a signal before any data was read
- EAGAIN - The file was marked for non-blocking I/O, and no data was ready to be read.

write

```
#include <unistd.h>
```

```
ssize_t write(int fd, const void *buff, size_t nbytes)
```

- Attempts to write *nbytes* of data to the object referenced by the descriptor *fd* from the buffer pointed to by *buff*.
- Upon successful completion, the number of bytes **actually written** is returned.
 - The number can be smaller than *nbytes*, even zero
- Otherwise -1 is returned and *errno* is set.
- “A successful return from write() does not make any guarantee that data has been committed to disk.”

write - errno

- EBADF - *fd* is not a valid descriptor or it is not open for writing.
- EPIPE - An attempt is made to write to a pipe that is not open for reading by any process.
- EFBIG - An attempt was made to write a file that exceeds the maximum file size.
- EINVAL - *fd* is attached to an object which is unsuitable for writing (such as keyboards).
- ENOSPC - There is no free space remaining on the file system containing the file.
- EDQUOT - The user's quota of disk blocks on the file system containing the file has been exhausted.
- EIO - An I/O error occurred while writing to the file system.
- EAGAIN - The file was marked for non-blocking I/O, and no data could be written immediately.

Example

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";

int main(void) {
    int fd;
    fd = creat("file.hole", S_IRUSR|S_IWUSR|IRGRP);
    if( fd < 0 ) {
        perror("creat error");
        exit(1);
    }
    if( write(fd, buf1, 10) != 10 ) {
        perror("buf1 write error");
        exit(1);
    }
    /* offset now = 10 */
    if( lseek(fd, 40, SEEK_SET) == -1 ) {
        perror("lseek error");
        exit(1);
    }
    /* offset now = 40 */
    if(write(fd, buf2, 10) != 10){
        perror("buf2 write error");
        exit(1);
    }
    /* offset now = 50 */
    exit(0);
}
```

Example – copying a file

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

enum {BUF_SIZE = 16};

int main(int argc, char* argv[])
{
    int fdread, fdwrite;
    unsigned int total_bytes = 0;
    ssize_t nbytes_read, nbytes_write;
    char buf[BUF_SIZE];

    if (argc != 3) {
        printf("Usage: %s source destination\n",
            argv[0]);
        exit(1);
    }

    fdread = open(argv[1], O_RDONLY);
    if (fdread < 0) {
        perror("Failed to open source file");
        exit(1);
    }

    fdwrite = creat(argv[2], S_IRWXU);
    if (fdwrite < 0) {
        perror("Failed to open destination file");
        exit(1);
    }

    do {
        nbytes_read = read(fdread, buf, BUF_SIZE);
        if (nbytes_read < 0) {
            perror("Failed to read from file");
            exit(1);
        }

        nbytes_write = write(fdwrite, buf,
            nbytes_read);
        if (nbytes_write < 0) {
            perror("Failed to write to file");
            exit(1);
        }
    } while(nbytes_read > 0);

    close(fdread);
    close(fdwrite);

    return 0;
}
```

dup2

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

- Duplicates an existing object descriptor and returns its value to the calling process.
- Causes the file descriptor *newfd* to refer to the same file as *oldfd*. The object referenced by the descriptor does not distinguish between *oldfd* and *newfd* in any way.
- If *newfd* refers to an open file, it is closed first
- Now *newfd* and *oldfd* file position pointer and flags.

dup2 - errno

- EBADF - *oldfd* isn't an open file descriptor, or *newfd* is out of the allowed range for file descriptors.
- EMFILE - Too many descriptors are active.
- Note:
If a *separate* pointer to the file is desired, a different object reference to the file must be obtained by issuing an additional `open()` call.

Dup2 - comments

- **dup2()** is most often used to redirect standard input or output.

~> `ls | grep "my"`

- **dup2(fd, 0)** - whenever the program tries to read from standard input, it will read from **fd**.
- **dup2(fd, 1)** - whenever the program tries to write to standard output, it will write to **fd**.
- After arranging the redirections, the desired program is run using *exec*

fcntl

```
#include <sys/types.h>  
#include <unistd.h>  
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd, int arg)
```

- manipulate file descriptors.
- *cmd* – the operation to perform
- *arg* – depends on the operation (not always required)

fcntl - cmd

- **F_DUPFD** - Returns a new descriptor as follows:
 - Lowest numbered available descriptor greater than or equal to *arg*.
 - New descriptor refers to the same object as *fd*.
 - New descriptor shares the same file offset.
 - Same access mode (read, write or read/write).
- This is different from `dup2` which uses exactly the descriptor specified.
- **F_GETFL** - Returns the current file status flags as set by `open()`.
 - Access mode can be extracted from AND'ing the return value
 - `return_value & O_ACCMODE`
- **F_SETFL** Set descriptor status flags to *arg*.
 - Sets the file status flags associated with *fd*.
 - Only `O_APPEND`, `O_NONBLOCK` and `O_ASYNC` may be set.

fcntl – example 1

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

int main( int argc, char *argv[] ){
    int accmode, val;
    if( argc != 2 ) {
        fprintf( stderr, "usage: %s <descriptor#>", argv[0] );
        exit(1);
    }
    val = fcntl(atoi(argv[1]), F_GETFL, 0);
    if ( val < 0 ) {
        perror( "fcntl error for fd" );
        exit( 1 );
    }
    accmode = val & O_ACCMODE;
    if( accmode == O_RDONLY )
        printf( "read only" );
    else if( accmode == O_WRONLY )
        printf( "write only" );
    else if( accmode == O_RDWR )
        printf( "read write" );
    else {
        fprintf( stderr, "unkown access mode" );
        exit(1);
    }
    if( val & O_APPEND )
        printf( ", append");
    if( val & O_NONBLOCK )
        printf(", nonblocking");
    if( val & O_SYNC )
        printf(", synchronous writes");
    putchar( '\n' );
    exit(0);
}
```

fcntl – example 2

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>

/* flags are file status flags to turn on */
void set_fl( int fd, int flags ){
    int val;
    val = fcntl( fd, F_GETFL, 0 );
    if (val < 0 ) {
        perror( "fcntl F_GETFL error" );
        exit( 1 );
    }
    val |= flags;      /* turn on flags */
    val = fcntl( fd, F_SETFL, val ) < 0
    if( val < 0 ) {
        perror( "fcntl F_SETFL error" );
        exit( 1 );
    }
}
```

Links

(hard links and soft links)

```
#include <unistd.h>
```

```
int link(const char *existingpath,  
         const char *newpath)
```

```
int symlink(const char *actualpath,  
            const char *newpath);
```

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

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

```
int rename (const char *oldname, const char *newname);
```

link – make a hard link

```
int link(const char *existingpath, const char *newpath)
```

- Makes a **hard link** to a file
- Atomically creates the specified directory entry (hard link) *newpath* with the attributes of the underlying object pointed at by *existingpath*.
- If the link is successful: the link count of the underlying object is incremented; *newpath* and *existingpath* share **equal access and rights** to the underlying object.
- If *existingpath* is removed, the file *newpath* is not deleted and the link count of the underlying object is decremented.

link() example

```
~> ls -l
total 8
-rwx----- 1 jphb 5804 Sep 25 15:44 mklink
-rw----- 1 jphb 98 Sep 25 15:43 mklink.c
-r----- 1 jphb 256 Sep 25 15:05 test
```

- Make a link in C:
`link("test", "new_name")`
- Make a link in the shell:
`~> ln test new_name`

```
~> ls -l
total 9
-rwx----- 1 jphb 5804 Sep 25 15:44 mklink
-rw----- 1 jphb 98 Sep 25 15:43 mklink.c
-r----- 2 jphb 256 Sep 25 15:05 new name
-r----- 2 jphb 256 Sep 25 15:05 test
```

symlink – make a soft (symbolic) link

```
int symlink(const char *actualpath,  
           const char *newpath);
```

- Makes a symbolic link to a file.
- To the normal user, a symbolic link behaves like a file,
 - but the underlying mechanism is different.
- Creates a special type of file whose contents are the *name* of the target file
- The existence of the file is not affected by the existence of symbolic links to it

link() example

```
~> ls -l
total 7
-rwx----- 1 jphb 5816 Sep 29 14:04 mklink
-rw----- 1 jphb 101 Sep 29 14:04 mklink.c
```

- Make a link in C:
`symlink("test", "new_name")`
- Make a link in the shell:
`~> ln -s test new_name`

```
~> ls -l
total 8
-rwx----- 1 jphb 5816 Sep 29 14:04 mklink
-rw----- 1 jphb 101 Sep 29 14:04 mklink.c
lrwxrwxrwx 1 jphb 4 Sep 29 14:04 new_name -> test
```

Does anyone see a problem here?

unlink

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

- Removes the link (soft or hard) named by *pathname* from its directory
- If a hard link, **decrements** the link count of the file which was referenced by the link.
- If that decrement reduces the link count of the file to zero, and no process has the file open, then all resources associated with the file are reclaimed.
- If one or more processes have the file open when the last link is removed, the link is removed, but the removal of the file is **delayed** until all references to it have been closed.

remove

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

- Removes the file or directory specified by *path*.
- If *path* specifies a directory, `remove(path)` is the equivalent of `rmdir(path)`. Otherwise, it is the equivalent of `unlink(path)`.

rename

```
int rename(const char *oldname, const char *newname);
```

- Causes the link named *oldname* to be renamed as *newname*.
- If the file *newname* exists, it is first removed.
 - The switch is done atomically
- Both *oldname* and *newname* must
 - be both directories or both non-directories
 - reside on the same file system
 - But they do not have to be on the same directory path
- If *oldname* is a symbolic link, the **symbolic link** is renamed, not the file or directory to which it points.

stat, fstat, lstat:

Get information about a file

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
int stat(const char *pathname, struct stat *buf)
```

```
int fstat(int fd, struct stat *buf)
```

```
int lstat(const char *pathname, struct stat *buf)
```

stat

```
int stat(const char *pathname, struct stat *buf)
```

- Obtains information about the file pointed to by *pathname*.
- Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

fstat

```
int fstat(int fd, struct stat *buf)
```

- Obtains the same information about an open file known by the file descriptor *fd*.

```
int lstat(const char *pathname, struct stat *buf)
```

- like **stat()** except that if the named file is a symbolic link, **lstat()** returns information about the link, while **stat()** returns information about the file the link references.

struct stat

```
Struct stat {
    mode_t st_mode;      /* file type and mode (type & permissions) */
    ino_t st_ino;        /* inode's number */
    dev_t st_dev;        /* device number (file system) */
    nlink_t st_nlink;    /* number of links */
    uid_t st_uid;        /* user ID of owner */
    gid_t st_gid;        /* group ID */
    off_t st_size;       /* size in bytes */
    time_t st_atime;     /* last access */
    time_t st_mtime;     /* last modified */
    time_t st_ctime;     /* last file status change */
    long st_blksize;     /* I/O block size */
    long st_blocks;      /* number of blocks allocated */
}
```

umask – user mask

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
mode_t umask(mode_t cmask)
```

- The umask command permission bits to **block** when a user creates directories and files
- The bits in the umask are **turned off** from the **mode** argument to open.
- Example: If the umask value is octal 022, the results in new files being created with permissions 0666 is $0666 \& \sim 0022 = 0644 = rw-r--r--$.

chmod

```
#include <sys/types.h>  
#include <sys/stat.h>
```

```
int chmod(const char *pathname, mode_t mode)
```

- Sets the file permission bits of the file specified by the pathname *pathname* to *mode*.
- User must be file owner to change mode

chown

```
#include <sys/types.h>
#include <unistd.h>

int chown(const char *pathname,
          uid_t owner,
          gid_t group);
```

- The owner ID and group ID of the file named by *pathname* is changed as specified by the arguments *owner* and *group*.
- The owner of a file may change the *group*.
- Changing the *owner* is usually allowed only to the superuser.