I/O OPERATIONS

UNIX Programming 2014 Fall by Euiseong Seo

Files

- Files that contain a stream of bytes are called regular files
- Regular files can be any of followings
  - ASCII text
  - Data
  - Executable code
  - Shell scripts
  - Something else

Files

- Files are composed of two parts
  - File contents (data part)
    - Linear sequence of bytes
    - Not terminated by any special character
  - Control part (inode)
    - File type
    - Location of data blocks
    - Permissions
    - Owner
    - Size of file

File Descriptor

- An integer number
- You must first get a file descriptor to access a file
- File descriptors are handles used to abstract and hide implementation from users
- File descriptors 0, 1 and 2 are special
  - 0: standard input
  - 1: standard output
  - 2: standard error
  - POSIX defines STDIN_FILENO, STDOUT_FILENO and STDERR_FILENO in <unistd.h> for portability

File Information

- stat(2) retrieves information from inode
- lstat(2) is similar to stat() but does not follow symbolic links
- Prototype

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

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

File Information

- stat information

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

File Information

- st_mode encodes permission and file type information

S_ISUID	0004000	set-user-ID bit
S_ISGID	0002000	set-group-ID bit (see below)
S_ISVTX	0001000	sticky bit (see below)
S_IRWXU	00700	mask for file owner permissions
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission

- <sys/stat.h> provides macros to determine file type or permission flag from st_mode

```
struct stat buf;

stat("file", &buf);
if(S_ISDIR(buf.st_mode))
    printf("It is a directory!\n");
```

Access Privilege

- A process's access privileges to a target file are determined by the following algorithm
  - if process effective uid matches file uid
    then process has file user access rights to file
  - else if process effective gid matches file gid
    then process has file group access rights to file
  - else
    process has file other access rights to file

Access Privilege Example

- This program is owned by “euiseong” , and has setuid bit

```
main()
{
    system("ls /home/euiseong ");
    system("cp /etc/passwd /home/euiseong");
    system("ls /home/euiseong");
    system("rm -f /home/euiseong/passwd");
    system("ls /home/euiseong");
}
```

Permissions

- Permission of a file is encoded in the bottom 12 bits of st_mode in inode structure
  - {Setuid,Setgid,Sticky}{UR,UW,UX}{GR,GW,GX}{OR,OW,OX}
- You can change permission of a file with chmod(2) system call
- If you want add or subtract permission from current settings, read them first

Permissions

- Example

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);

#include <sys/stat.h>
main()
{
    struct stat buf;

    stat("filename", &buf);
    buf.st_mode |= S_IWUSR | S_ISGID; /* chmod u+w g+s */
    buf.st_mode &= ~(S_IWGRP | S_IWOTH); /*  chmod g-w o-w */
    if(chmod("filename", buf.st_mode & 07777) == -1)
        perror("chmod");
}
```

Opening Files

- open(2) routine checks permissions and returns a file descriptor if access is allowed
- open() can also create new regular files, or truncate existing files
- Prototype

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

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

Opening Files

- flags is a bitwise combination of following macros
  - O_RDONLY – read only
  - O_WRONLY – write only
  - O_RDWR – read and write
  - O_CREAT – create file and use mode as permission
  - O_EXCL – exclusive create
  - O_APPEND – append
  - O_NONBLOCK – non-blocking I/O
  - O_TRUNC – truncate file

User Mask

- mode passed to open() to create a file is used after being processed with user mask (also called file creation mode mask)
- User mask specifies which permissions will not be granted
- Default mask is 022 (therefore, new file has 755)
- You can change user mask with umask(2) or umask(1)

User Mask Example

```
#include <fcntl.h>

main()
{
    mode_t oldmask;
    int fd;

    /* turn off group write prohibition temporarily */
    oldmask = umask (002);
    fd = open("testfile1", O_WRONLY|O_CREAT, 0666);
    umask(oldmask);
    fd = open("testfile2", O_WRONLY|O_CREAT, 0666);
}
```

```
-rw-rw-r--    1 root       staff       0 Nov   1 09:20 testfile1
-rw-r--r--    1 root       staff       0 Nov   1 09:20 testfile2
```

Open File Table

Process Open-File Table
of PID 45

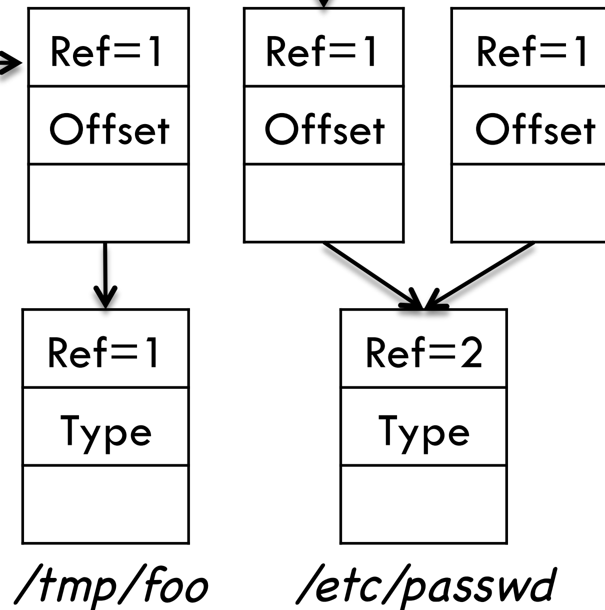
0	stdin
1	stdout
2	stderr
3	
4	

Global File Table

Process Open-File Table
of PID 48

0	stdin
1	stdout
2	stderr
3	
4	

Inode Table

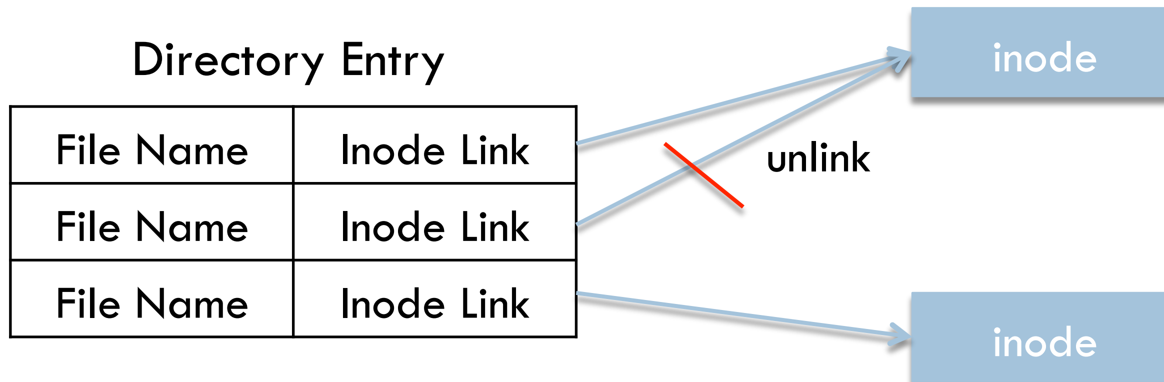


Open and FD Number

- Open starts looking for the lowest available descriptor starting at zero
- What happen when we close STDIN and open a file?
- This fact will be used to implement redirection for your shell

Remove a File

- unlink(2) deletes a file (actually, a link between directory entry and inode)
- unlink() decreases file reference count in inode



Remove a File

- Prototype

```
#include <stdio.h>
int remove(const char *path); // POSIX

#include <unistd.h>
int unlink(const char *path); // ANSI C
```

- Space is released when inode reference count becomes zero and no process uses the file

Read and Write

- Reading from a file is performed using read(2)
- Writing is performed using write(2)
- Prototype

```
ssize_t read(int fd, void *buf, fize_t nbytes);
ssize_t write(int fd, const void *buf, size_t nbytes);
```

Seeking

- To change offset of next read or write, use lseek(2)
- Prototype

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

- whence
  - SEEK_SET: offset is from beginning of file
  - SEEK_CUR: offset is relative to current position
  - SEEK_END: offset is relative to end of file
- You can seek backward by specifying a negative offset from current position

Closing

- When you are done with a file, you should close it
- Closing a FD may free up system resources
- All of the processes open files are automatically closed when process is terminated
- Prototype

```
#include <unistd.h>

int close(int fildes);
```

Synchronization to Disk

- File data is normally buffered in memory for performance
- Unexpected crash may blow out buffered data before they are written to disks
- Two ways to prevent this
  - Open file with O_SYNC flag
  - Use fsync(2)
- Prototype

```
#include <unistd.h>
int fsync(int fildes);
```

Checking Accessability

- open() checks permissions against EUID and EGUID
- If you need to check against real UID, use access(2)
- Prototype

```
#include <unistd.h>
int access(const char *pathname, int mode);
```

- Mode
  - R_OK – test for read permission
  - W_OK – test for write permission
  - X_OK – test for execute or search permission
  - F_OK – test for existence of file

Checking Accessability

- Example

```
if(access(".rcfile", F_OK) == -1 && errno == ENOENT) {
    printf("Default startup file does not exist.\n");
    printf("Using default .rcfile.\n");
    create_defaults(".rcfile");
}
```

Standard I/O and System-Level I/O

- What we've learned are for system-level I/O
- fopen and f-series functions are for standard I/O
- Standard I/O is built upon system-level I/O
- Standard I/O manages user-level buffer
  - Standard I/O = system-level I/O + buffering + programmer-friendly interface
  - For performance and efficiency
  - Users cannot control when actual I/O operations occur

Related Standard I/O Calls

- You can get a FILE pointer out of an open FD
- Never use FD again, but do not close it too
- Prototype

```
#include <stdio.h>
FILE *fdopen(int fd, const char *type);
int fileno(FILE *fp);
```

Reopening FP

- There are three special file pointers
  - stdin, stdout and stderr
  - E.g.) fprintf(stderr, “Error!\n”);
- freopen(3) closes an existing FP stream, opens a file and associates this file stream with FP

```
#include <stdio.h>
main()
{
    if(freopen("/tmp/foo3", "w", stdout) != stdout)
        {fprintf(stderr, "ERROR"); exit(1); }
    printf("Hi Folks!\n");
}
```

Temporary Files

- If you need a temporary file, these will be useful
- Prototype

```
#include <stdio.h>
char *tmpnam(char *s);
FILE *tmpfile(void);
```

Simple File Control

- fcntl(2) provides many file operations by manipulating file descriptors
- Prototype

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */ );
```

- Example command
  - F_GETFL – get status flags
  - F_SETFL – set status flags