EE309 Advanced Programming Techniques for EE

Lecture 5: Files and Directories

INSU YUN (윤인수)

School of Electrical Engineering, KAIST

[Some Lecture Slides Based on Prof. Shin SEUNGWON 2020]

Today’s lecture

- Learn APIs for files and directories

Standard I/O

Standard I/O Functions

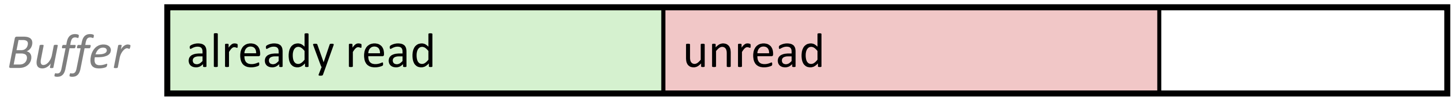
- The C standard library (`libc.so`) contains a collection of higher-level *standard I/O* functions

- Examples of standard I/O functions:
  - Opening and closing files (**`fopen`** and **`fclose`**)
  - Reading and writing bytes (**`fread`** and **`fwrite`**)
  - Reading and writing text lines (**`fgets`** and **`fputs`**)
  - Formatted reading and writing (**`fscanf`** and **`fprintf`**)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

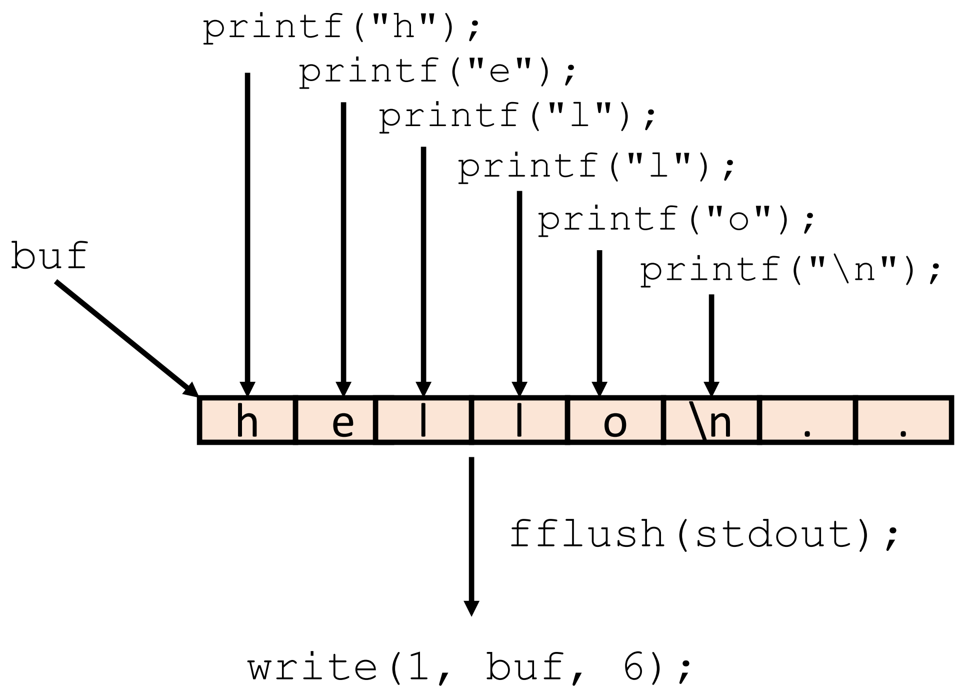
Buffered I/O: Motivation

- Applications often read/write one character at a time
  - `getc, putc, ungetc`
  - `gets, fgets`
    - Read line of text one character at a time, stopping at newline
- Implementing as Unix I/O calls expensive
  - `read` and `write` require Unix kernel calls
    - > 10,000 clock cycles
- Solution: Buffered read
  - Use Unix `read` to grab block of bytes
  - User input functions take one byte at a time from buffer
    - Refill buffer when empty



Buffering in Standard I/O

- Standard I/O functions use buffered I/O



- Buffer flushed to output fd on “₩n”, call to `fflush` or `exit`, or return from `main`.

Standard I/O Buffering in Action

- You can see this buffering in action for yourself, using the always fascinating Linux `strace` program:

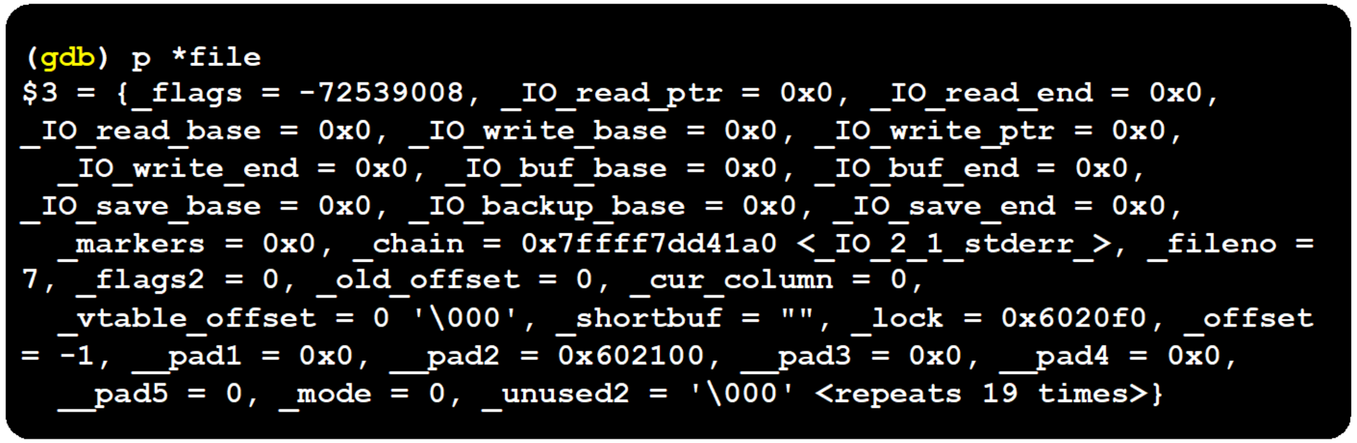
```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6)                = 6
...
exit_group(0)                         = ?
```

Standard I/O: In detail

FILE* based I/O

- One of the basic ways to manage input and output is to use the FILE set of functions provided by libc.
  - The FILE structure is a set of data items that are created to manage input and output for the programmer.
  - An abstraction of "high level" reading and writing files that avoids some of the details of programming.
  - Almost always used for reading and writing ASCII data



```
(gdb) p *file
$3 = {_flags = -72539008, _IO_read_ptr = 0x0, _IO_read_end = 0x0,
_IO_read_base = 0x0, _IO_write_base = 0x0, _IO_write_ptr = 0x0,
  _IO_write_end = 0x0, _IO_buf_base = 0x0, _IO_buf_end = 0x0,
_IO_save_base = 0x0, _IO_backup_base = 0x0, _IO_save_end = 0x0,
  _markers = 0x0, _chain = 0x7ffff7dd41a0 <_IO_2_1_stderr_>, _fileno =
7, _flags2 = 0, _old_offset = 0, _cur_column = 0,
  _vtable_offset = 0 '\000', _shortbuf = "", _lock = 0x6020f0, _offset
= -1, __pad1 = 0x0, __pad2 = 0x602100, __pad3 = 0x0, __pad4 = 0x0,
  __pad5 = 0, _mode = 0, _unused2 = '\000' <repeats 19 times>}
```

```c
struct _IO_FILE {
  int _flags;           /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags

  /* The following pointers correspond to the C++ streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
#if 0
  int _blksize;
#else
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */

#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

fopen()

- The fopen function opens a file for IO and returns a pointer to a FILE* structure:
  - FILE *fopen(const char *path, const char *mode);
- Where,
  - path is a string containing the absolute or relative path to the file to be opened.
  - mode is a string describing the ways the file will be used
  - For example,
    FILE *file = fopen(filename, "r+");
  - Returns a pointer to FILE* if successful, NULL otherwise
    - You don’t have to allocate or deallocate the FILE* structure

fopen() mode

- “r” - Open text file for reading. The stream is positioned at the beginning of the file.
- “r+”-Open for reading and writing. The stream is positioned at the beginning of the file.
- “w” - Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
- “w+” - Open for reading and writing. The file is created if it does not exist, otherwise it is truncated.
- “a” Open for appending (writing at end of file). The file is created if it does not exist.
- “a+” Open for reading and appending (writing at end of file). The file is created if it does not exist.

Reading the file

- There are two dominant ways to read the file, fscanf and fgets
  - fscanf reads the data from the file just like scanf, just reading and writing, e.g.,

```
if ( fscanf( file, "%d %d %d\n", &x, &y, &z ) == 3 ) {
  printf( "Read coordinates [%d,%d,%d]\n", x, y, z );
}
```

  - fgets reads the a line of text from the file, e.g.,

```
if ( fgets(str,128,file) != NULL  ) {
  printf( "Read line [%s]\n", str );
}
```

Writing the file

- There are two dominant ways to write the file, fprintf and fputs
  - fprintf writes the data to the file just like printf, just reading and writing, e.g.,

```
fprintf( file, "%d %d %d\n", x, y, z );
```

  - fputs writes the a line of text to the file, e.g.,

```
if ( fputs(str,file) != NULL  ) {
  printf( "wrote line [%s]\n", str );
}
```

fflush()

- FILE*-based IO is buffered
  - fflush attempts to reset/the flush state
    - int fflush(FILE *stream);

    - FILE*-based writes are buffered, so there may be data written, but not yet pushed to the OS/disk.
      - fflush() forces a write of all buffered data
    - FILE*-based reads are buffered, so the current data (in the process space) may not be current
      - fflush() discards buffered data from the underlying file

- If the stream argument is NULL, fflush() flushes *all* open output streams

fclose()

- fclose() closes the file and releases the memory associated with the FILE* structure.

```
fclose( file );
file = NULL;
```

Note: `fclose` implicitly flushes the data to storage.

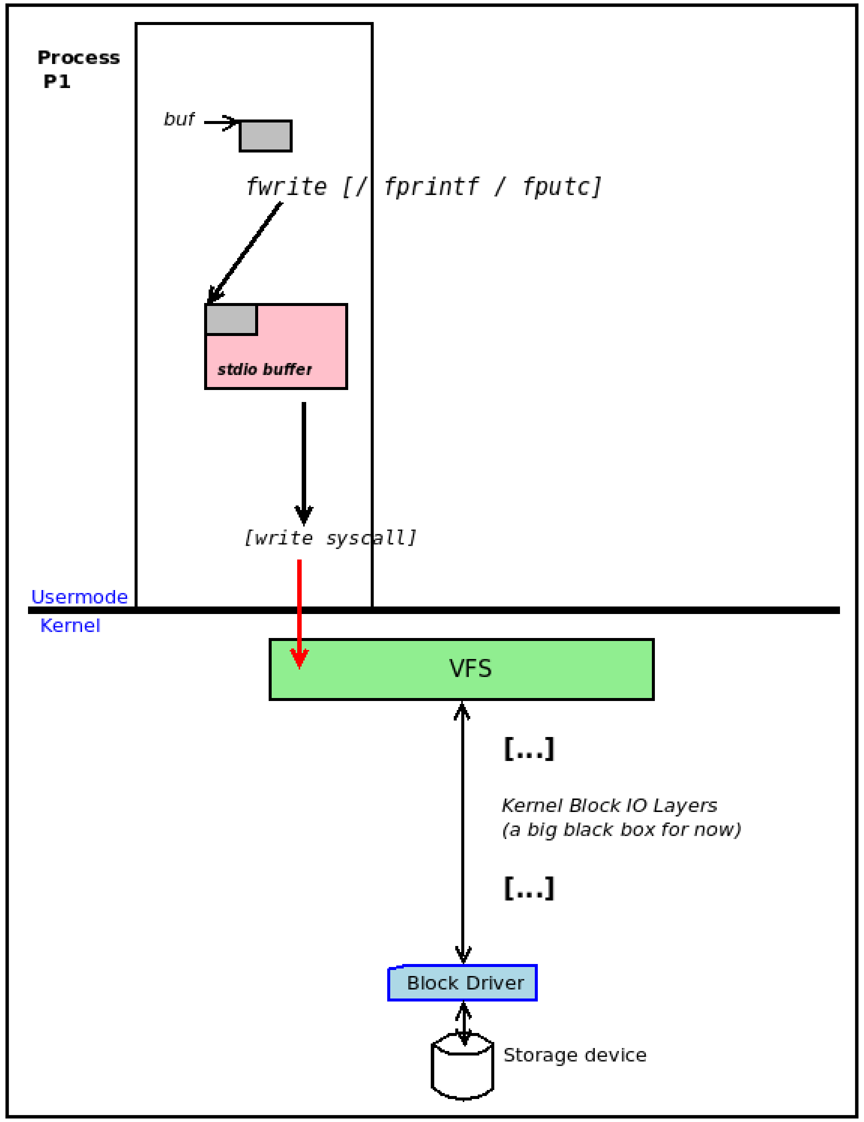
Example program

```c
int show_fopen( void ) {

    // Setup variables
    int x, y, z;
    FILE *file;
    char *filename = "/tmp/fopen.dat", str[128];
    file = fopen( filename, "r+" );

    // open for reading and writing
    if ( file == NULL ) {
        fprintf( stderr, "fopen() failed, error=%s\n", strerror(errno) );
        return( -1 );
    }

    // Read until you reach the end
    while ( !feof(file) ) {
        if ( fscanf( file, "%d %d %d\n", &x, &y, &z ) == 3 ) {
                printf( "Read coordinates [%d,%d,%d]\n", x, y, z );
        }
        if ( !feof(file) ) {
            fgets(str,128,file); // Need to get end of previous line
            if ( fgets(str,128,file) != NULL  ) {
                printf( "Read line [%s]\n", str );
            }
        }
    }
    // Now add some new coordinates
    x = 21;
    y = 34;
    z = 98;
    fprintf( file, "%d %d %d\n", x, y, z );
    printf( "Wrote %d %d %d\n", x, y, z );
    if ( fputs(str,file) >= 0 ) {
        printf( "wrote line [%s]\n", str );
    }
    fflush( file );

    // Close the file and return
    fclose( file );
    return( 0 );
}
```

```
$ cat /tmp/fopen.dat
1 2 3
4 5 6
11 12 14
16 17 23
$ ./io
This is cmpsc311, IO example
Read coordinates [1,2,3]
Read line [11 12 14
]
Read coordinates [16,17,23]
Wrote 21 34 98
wrote line [11 12 14
]
$ cat /tmp/fopen.dat
1 2 3
4 5 6
11 12 14
16 17 23
21 34 98
11 12 14
$
```

fopen() vs. open()

- Key differences between fopen and open
  - fopen provides you with buffering IO that may or may not turn out to be a faster than what you're doing with open.
  - fopen does line ending translation if the file is not opened in *binary mode*, which can be very helpful if your program is ever ported to a non-Unix environment.
  - A FILE * gives you the ability to use fscanf and other stdio functions that parse out data and support formatted output.
- When to use (IMO)
  - use FILE* style I/O
    - High level abstraction is required (porting), for ASCII processing
  - file handle I/O
    - If you deeply understand how to handle IO, for binary data processing


Process
P1
buf
fwrite [/ fprintf / fputc]
stdio buffer
[write syscall]
Usermode
Kernel
VFS
[...]
Kernel Block IO Layers
(a big black box for now)
[...]
Block Driver
Storage device

- Each of the styles of I/O requires a different set of include files
  - FILE* requires:

```
#include <stdio.h>
```

  - file handle I/O requires:

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

Buffered I/O

- When the system is buffering
  - It may read more that requested in the expectation you will
- read more later (read buffering)
  - it may not commit all bytes to the target (write buffering)

Unbuffered I/O ?

Metadata

stat(), lstat()

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

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

All return: 0 if OK, 1 on error
```

- Returns a structure of information about the named file
- `lstat()` vs `stat()`: Returns information about the symbolic link, not the file referenced by the symbolic link
  - Explain the symbolic link later

```c
struct stat {
  mode_t    st_mode;      /* file type & mode (permissions) */
  ino_t     st_ino;       /* i-node number (serial number) */
  dev_t     st_dev;       /* device number (file system) */
  dev_t     st_rdev;      /* device number for special files */
  nlink_t   st_nlink;     /* number of links */
  uid_t     st_uid;       /* user ID of owner */
  gid_t     st_gid;       /* group ID of owner */
  off_t     st_size;      /* size in bytes, for regular files */
  time_t    st_atime;     /* time of last access */
  time_t    st_mtime;     /* time of last modification */
  time_t    st_ctime;     /* time of last file status change */
  blksize_t st_blksize;   /* best I/O block size */
  blkcnt_t  st_blocks;    /* number of disk blocks allocated */
};
```

File types

- We've talked about two different types of files so far: regular files and directories.

- Most files on a UNIX system are either regular files or directories, but there are additional types of files. The types are:
  - Regular file
  - Directory file
  - Socket: A type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host.
  - Symbolic link. A type of file that points to another file (Later)
  - …

Example

```
int main(int argc, char *argv[]) {
    int          i;
    struct stat buf;
    char         *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            perror("lstat error");
            continue;
         }
         if (S_ISREG(buf.st_mode))
            ptr = "regular";
         else if (S_ISDIR(buf.st_mode))
            ptr = "directory";
         else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
         else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
         else
            ptr = "** unknown mode **";
         printf("%s\n", ptr);
  }
}
```

Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISLNK()	symbolic link
S_ISSOCK()	socket
…	…

```
$ sudo ./lstat /etc/passwd \
      /etc \
      /var/run/mysqld/mysqld.sock \
      /dev/stdin
/etc/passwd: regular
/etc: directory
/var/run/mysqld/mysqld.sock: socket
/dev/stdin: symbolic link
```

Q: What happens if I change lstat to stat?

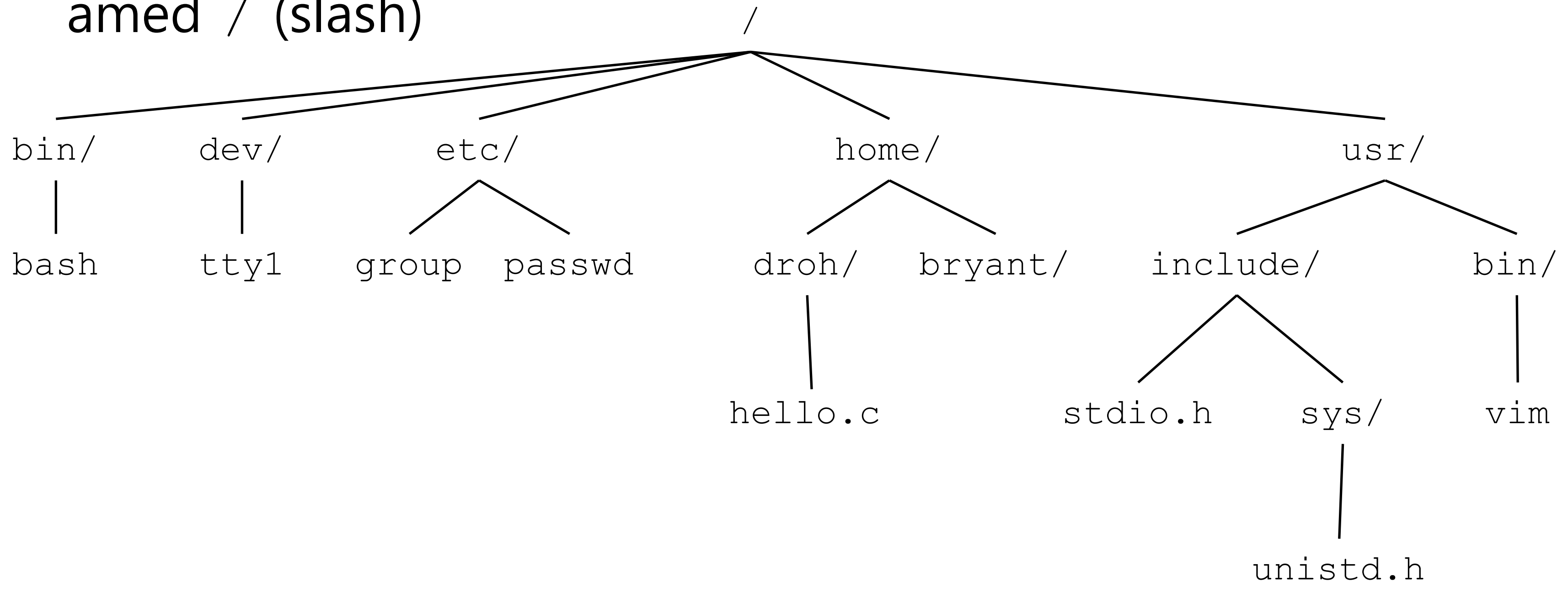
Directories

Directories

- Directory consists of an array of *links*
  - Each link maps a *filenam*e to a file
- Each directory contains at least two entries
  - `.` (dot) is a link to itself
  - `..` (dot dot) is a link to *the parent directory* in the *directory hierarch y* (next slide)
- Commands for manipulating directories
  - `mkdir`: create empty directory
  - `ls`: view directory contents
  - `rmdir`: delete empty directory

Directory Hierarchy

- All files are organized as a hierarchy anchored by root directory n amed `/` (slash)



- Kernel maintains *current working directory (cwd)* for each process
  - Modified using the `cd` command

Reading Directories

```
#include <dirent.h>

DIR *opendir(const char *pathname);
// Returns: pointer if OK, NULL on error

struct dirent *readdir(DIR *dp);
// Returns: pointer if OK, NULL at end of d
irectory or error

int closedir(DIR *dp);
// Returns: 0 if OK, 1 on error
```

```c
struct dirent {
 ino_t          d_ino;       /* Inode number */
 off_t          d_off;       /* Not an offset; see below */
 unsigned short d_reclen;    /* Length of this record */
 unsigned char  d_type;      /* Type of file; not supported
                                by all filesystem types */
 char           d_name[256]; /* Null-terminated filename */
};
```

```c
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  DIR *dir;
  struct dirent *ent;
  if ((dir = opendir (argv[1])) != NULL) {
    /* print all the files and directories wit
hin directory */
    while ((ent = readdir (dir)) != NULL) {
      printf ("%s ", ent->d_name);
    }
    printf("\n");
    closedir (dir);
  } else {
    /* could not open directory */
    perror ("");
    return EXIT_FAILURE;
  }
}
```

```
$ ./listdir /
home srv etc opt root Docker li
b mnt usr media lib64 sys dev s
bin boot bin run lib32 libx32 i
nit proc snap tmp var lost+foun
d .. .
```

```c
#include <dirent.h>

int scandir(const char *restrict dirp,
           struct dirent ***restrict namelist,
           int (*filter)(const struct dirent *),
           int (*compar)(const struct dirent **,
                         const struct dirent **));

int alphasort(const struct dirent **a, const struct dirent **b);
```

```c
#define _DEFAULT_SOURCE
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
   struct dirent **namelist;
   int n;

   n = scandir(".", &namelist, NULL, alphasort);
   if (n == -1) {
       perror("scandir");
       exit(EXIT_FAILURE);
   }

   while (n--) {
       printf("%s\n", namelist[n]->d_name);
       free(namelist[n]);
   }
   free(namelist);

   exit(EXIT_SUCCESS);
}
```

Security

Access control

- The UNIX filesystem implements *discretionary access control* through file permissions set by user
  - The permissions are set at the discretion of the user

- Every file in the file system has a set of bits which determine who has assess to the file
  - User: the owner is typically the creator of the file, and the entity in control of the access control policy
  - Group: a set of users on the system setup by the admin
  - Other: the set of everyone on the system

- Note: this can be overridden by the “root” user

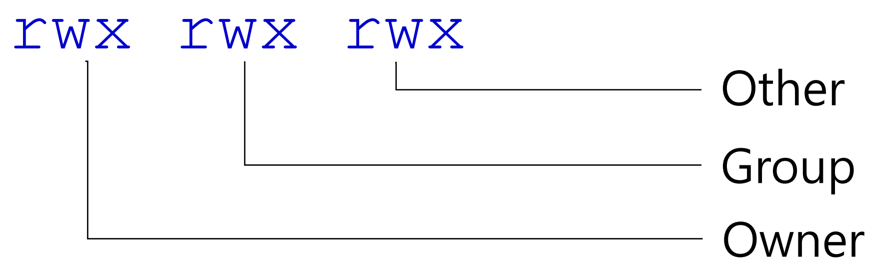
Unix/Linux file system permissions

- There are three permissions in the UNIX filesystem
  - READ: allows the subject (process) to read the contents of the file
  - WRITE: allows the subject (process) to alter the contents of the file
  - EXECUTE: allows the subject (process) to execute the contents of the file (e.g., shell program, executable)

- For directory
  - READ: allows the subject (process) to list the files in the directory
  - WRITE: allows the subject (process) to write (e.g., create, rename, delete, modify) files in the directory
  - EXECUTE: allows the subject (process) to access files in the directory
    - e.g., to create (or delete) a file, you also need executable permission

Unix/Linux Access Policy

- Really, this is a bit string encoding an access policy:



- And a policy is encoded as "r", "w", "x" if enabled, and "-" if not, e.g.,

```
$ ls -l .
total 28
-rw-r--r-- 1 insu insu     0 Aug 14 20:20 fopen.dat
-rwxr-xr-x 1 insu insu 16464 Aug 14 20:20 hello
-rw-r--r-- 1 insu insu    16 Aug 14 20:20 hello.c
-rwxr-xr-x 1 insu insu    12 Aug 14 20:20 hello.sh
```

- Says us and write, and wo

The nine file access permission bits, from <sys/stat.h>

st_mode mask	**Meaning**
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

```c
int main(int argc, char *argv[]) {
    int         i;
    struct stat buf;
    char        *ptr;

    for (i = 1; i < argc; i++) {
      printf("%s: ", argv[i]);
      if (lstat(argv[i], &buf) < 0) {
        perror("lstat error");
        continue;
      }

      char str[] = "---------";
      mode_t mode = buf.st_mode;

      if ( mode & S_IRUSR ) str[0] = 'r';
      if ( mode & S_IWUSR ) str[1] = 'w';
      if ( mode & S_IXUSR ) str[2] = 'x';

      if ( mode & S_IRGRP ) str[3] = 'r';
      if ( mode & S_IWGRP ) str[4] = 'w';
      if ( mode & S_IXGRP ) str[5] = 'x';

      if ( mode & S_IROTH ) str[6] = 'r';
      if ( mode & S_IWOTH ) str[7] = 'w';
      if ( mode & S_IXOTH ) str[8] = 'x';

      printf("%s\n", str);
    }
}
```

```
$ sudo ./permission /etc/passwd \
        /etc \
        /var/run/mysqld/mysqld.sock \
        /dev/stdin
/etc/passwd: rw-r--r--
/etc: rwxr-xr-x
/var/run/mysqld/mysqld.sock: rwxrwxrwx
/dev/stdin: rwxrwxrwx
```

User IDs and Group IDs

- Every process has four or more IDs associated with it

- Real user id (uid), Real group ID (gid)
  - who we really are
  - determined when we log in

- Effective user id (euid), Effective group ID (egid)
  - used for file access permission checks

setuid & setgid

- Every file has an owner and a group owner.
  - the owner: st_uid of the stat structure
  - the group owner: st_gid

- When we execute a program file,
  - Usually, the effective user ID == the real user ID
  - setuid & setgid: Special flags in the file's mode
    - If set, set the effective user ID (group ID) of the process to the owner (group) of the file
    - rw**s**rw**s**rwx: a bit string encoding for setuid & setgid
    - S_ISUID, S_ISGID: mask for setuid & setgid

How permission checking works

- If the effective user ID of the process is 0 (the superuser), access is allowed.
- If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the file), access is allowed
- If the effective group ID of the process (or one of the supplementary group IDs of the process) equals the group ID of the file, access is allowed
- If the appropriate other access permission bit is set, access is allowed.
- Otherwise, permission is denied.

Quiz

```
$ id
uid=1002(alice) gid=1003(alice) groups=1003(alice)
```

```
# Can I read these files?
$ ls -l
total 16
-rw-rw-r-- 1 alice alice 12 Aug 14 21:45 file1
-rw-rw-r-- 1 root  alice 12 Aug 14 21:42 file2
-rw-rw-r-- 1 root  root  12 Aug 14 21:45 file3
-r--r----- 1 root  root  12 Aug 14 21:46 file4
```

```
# Can I read file4 using cat?
$ ls -l
total 60
-rwxr-xr-x 1 alice alice 43416 Aug 14 21:47 cat
...
-r--r----- 1 root  root     12 Aug 14 21:46 file4
```

```
# Can I read file4 using cat?
$ ls -l
total 60
-rwsr-xr-x 1 alice alice 43416 Aug 14 21:47 cat
...
-r--r----- 1 root  root     12 Aug 14 21:46 file4
```

```
# Can I read file4 using cat?
$ ls -l
total 60
-rwsr-xr-x 1 root alice 43416 Aug 14 21:47 cat
...
-r--r----- 1 root  root     12 Aug 14 21:46 file4
```

`$ man chmod`

- Change file mode bits (i.e., permissions)
- **`chmod`** *[OPTION]... OCTAL-MODE FILE...*

- e.g.,
  - `chmod 755 hello.txt`
    Change hello.txt's permission to rwxr-xr-x
    (Octal mode: `r` = 4, `w` = 2, `x` = 1)

  - `chmod 4755 hello.txt`
    Change hello.txt's permission to rwsr-x-r-x
    (Special permissions: setuid = 4, setgid = 2, sticky bit = 1)

`$ man chown`

- Change file owner and group
- **chown** [*OPTION*]... [*OWNER*][*:*[*GROUP*]] *FILE*...

- e.g.,
  - `chown root hello.txt`
    Change the owner of hello.txt to “root”
  - `chown root:staff hello.txt`
    Likewise, but also change its group to “staff”

Symbolic link

- A symbolic link is an indirect pointer to a file
  - e.g., .lnk file in Windows
- You can create it using `ln` command
  - e.g., `ln -s [src] [dst]`
- Interesting property regarding security: You can create symbolic link even you don't have enough permission for source
  - e.g., You can make symbolic link for a file even you cannot read the file, or the file has setuid permission

Quiz

```
if(!access(file,W_OK)) {
  f = fopen(file,"w+");
  operate(f);
  ...
}
else {
  fprintf(stderr,"Unable to open file %s.\n",file);
}
```

- Let's assume that this is a setu
- NOTE: access() is a function th   on with an original user (not root).
- Can I write a file that only root can do?

Yes. That's what we say time-of-check to time-of-use (TOCTOU)

f*, *at functions

- There are multiple variant functions that prevent TOCTOU
  - openat()
  - faccessat()
  - fstat()
  - fchown(),
  - …

- You should use them for protecting from TOCTOU
  - In the previous example, open a file first, then use fstat to check permission manually