

# Advanced Unix Programming

## Module 04

Raju Alluri

askraju @ spurthi.com

# Advanced Unix Programming: Module 4

- Prerequisites
  - System Calls and Library Functions
  - Device Drivers
- Files & Directories
  - File creation related
  - File status check related

# Man Pages: Sections & their significance

- Man pages come as different sections
  - Section 1 contains administrative commands (mostly executed by superuser)
  - Section 2 contains system calls
  - Section 3 contains programming APIs
  - Section 5 contains configuration files
- Man pages for same name from different sections
  - “man stat” gives section 1 man page
  - “man -s 2 stat” (Solaris) or “man -s 2 stat” (Linux) gives section 2 man page

# System Calls

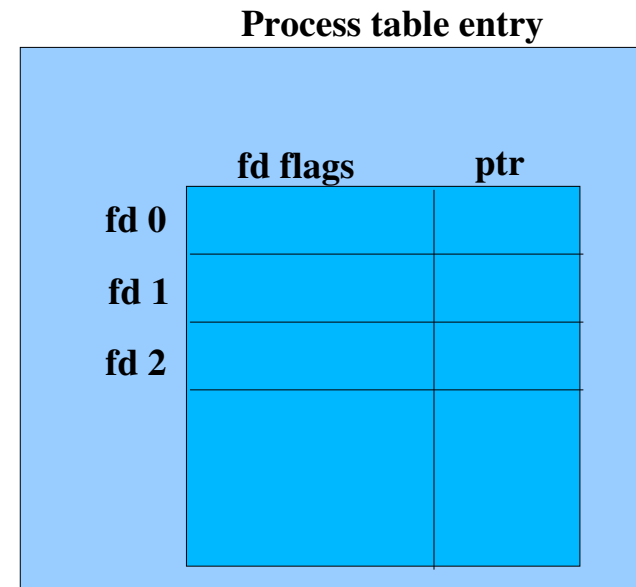
- System calls are the programmatic entry into the Kernel
- Typically they have C language programming APIs.
- They are documented as part of the section 2 of man pages
- Its very difficult to implement or impractical to use alternate implementations of these

# Library Functions

- Library functions are convenience functions that do some specific tasks
- They may internally call system calls.
- They are documented as part of the section 3 of man pages
- Its practicable to develop and use alternate versions
- There are two ways you can link the library implementations
  - Static, Dynamic
  - Using “ldd” to find out the library dependencies

# Unix File Structure, Process Table

- Each file is associated with a number called file descriptor (fd for short)
  - The file descriptor signifies the process table entry for that file
  - The process table entry contains
    - The fd\_flags
    - Pointer to a file table entry



# Unix File Structure, File Table

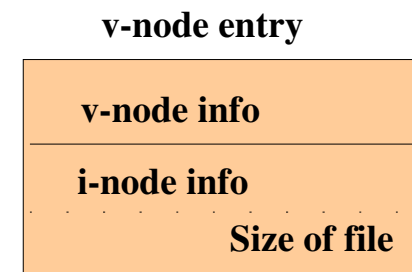
- Each File table entry will have
  - File status flags
  - Current file offset
  - Pointer to v-node table entry

**File table entry**

<b>File status flags</b>
<b>Current file offset</b>
<b>v-node ptr</b>

# Unix File Structure, v-node Table

- Each v-node table entry will have
  - v-node information
    - Type of file
    - Pointers to functions that operate on file etc.
  - i-node information
    - Owner of the file
    - Device information
    - Current file size



# Opening a file – open() system call

- Opens the given filename and returns the file descriptor

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

- Opens the file indicated by path
- oflag can be one of the following, and several other options OR-ed
  - O\_RDONLY: read only
  - O\_WRONLY: write only
  - O\_RDWR: for both read and write
- Other options:
  - O\_APPEND, O\_CREAT, O\_TRUNC, O\_EXCL, O\_SYNC etc

# Creating a file – creat() system call

- Creates the given filename for writing and returns the file descriptor

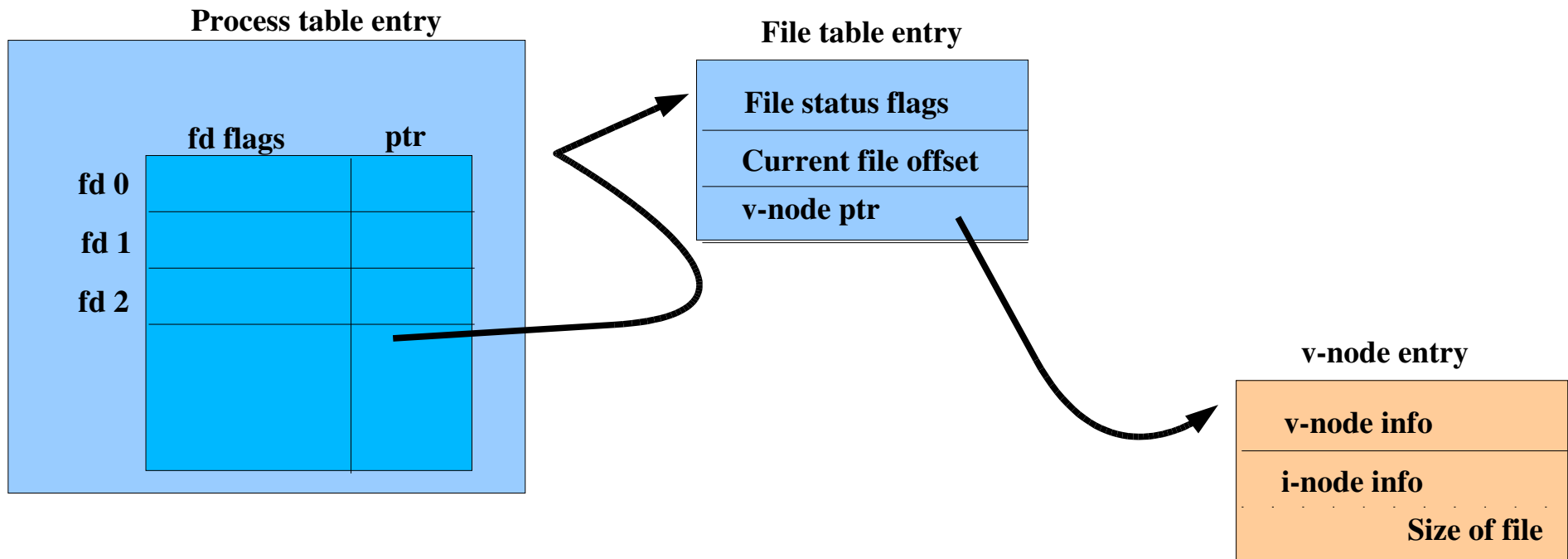
```
int creat(const char *path, mode_t mode);
```

- Creates the file indicated by path
- Functionally same as

```
open(path, O_WRONLY | O_CREAT | O_TRUNC,  
mode);
```

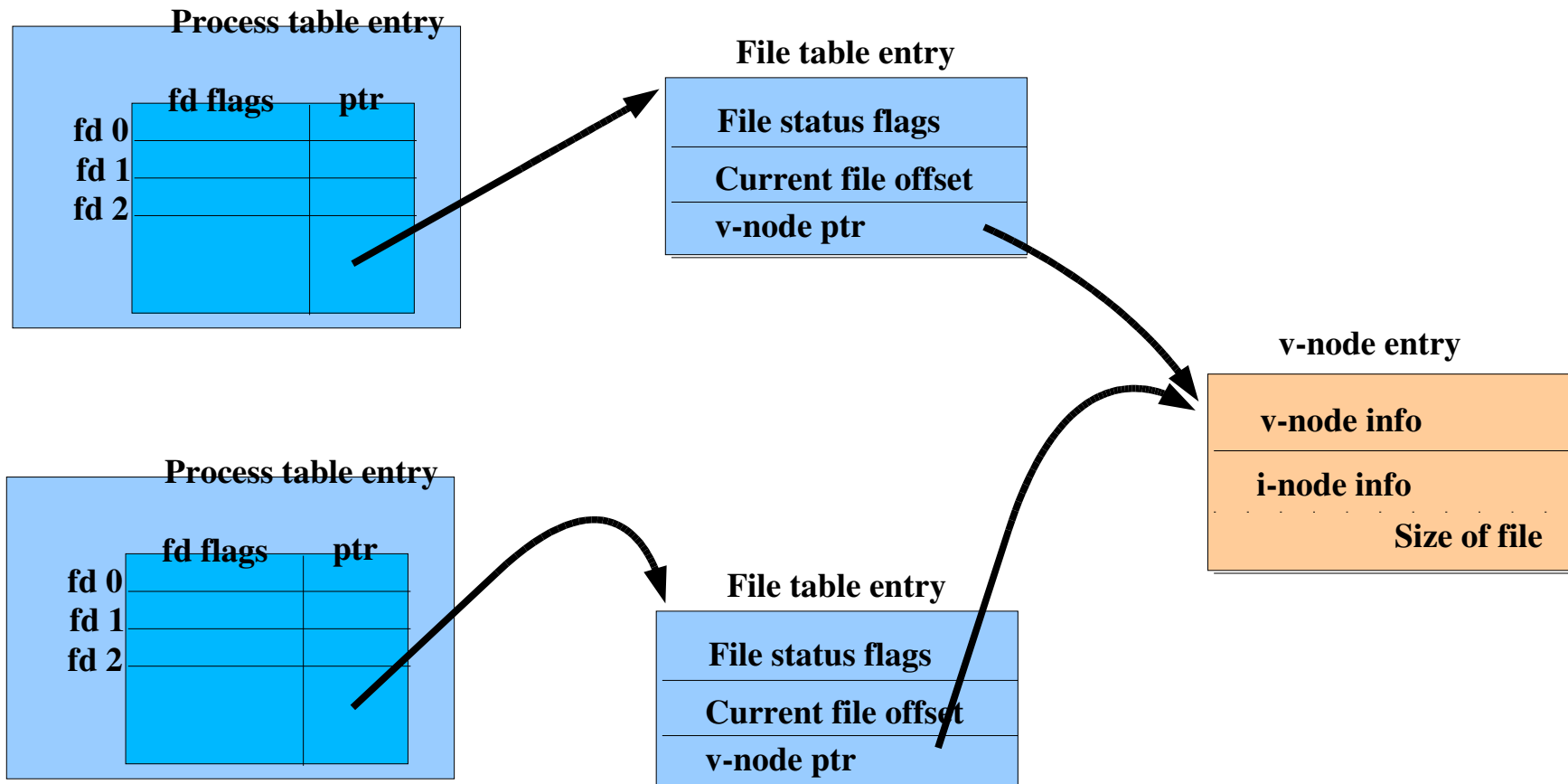
# Result of a open/create system call

- Creates a process table entry, file table entry and v-node entry



# Sharing files between processes

- There are separate process table and file table entries, but they point to same v-node entry



# Closing a file – close() system call

- Closes the file corresponding to the file descriptor

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

# Going to a location in a file: lseek()

- Set the current file offset to the desired value

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

– Whence indicates the direction of the set for offset

- SEEK\_SET: from the beginning of the file
- SEEK\_CUR: from the current location of the file
- SEEK\_END: from the end fo the file, where offset can be +ve or -ve

# Reading from a file: read()

- Read desired number of bytes from a file to a given buffer

```
ssize_t read(int fd, void *buf, size_t n);
```

- Read n number of bytes from file designated by fd and store it in buf

# Writing to a file: write()

- Write desired number of bytes from the given buffer to a file

```
ssize_t write(int fd, const void *buf,  
             size_t n);
```

- Write n number of bytes from buf to file designated by fd

# Duplicating file descriptors: dup, dup2

- Duplicate the given file descriptors

```
int dup(int fd);
```

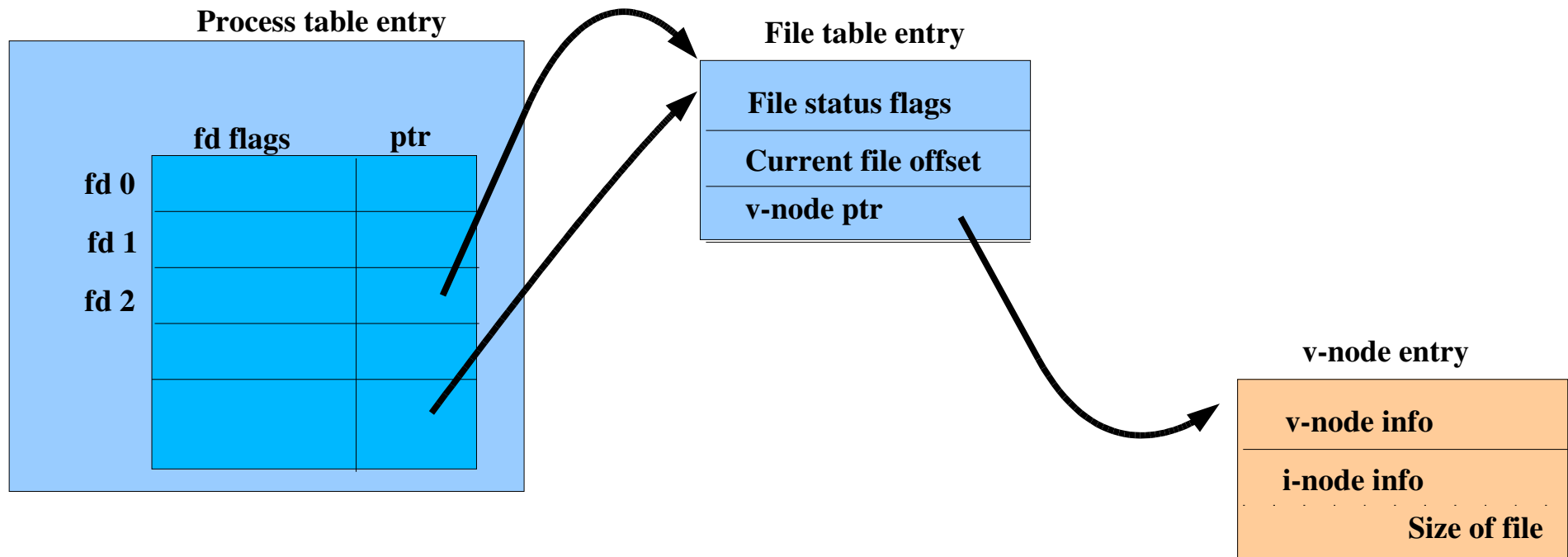
- Duplicate the fd and return the new file descriptor

```
int dup2(int fd, int fd2);
```

- Duplicate fd to fd2, and return the value
  - If fd2 is same as fd, return fd2
  - If fd2 is open, close it first
  - Now duplicate fd to fd2, and return fd2

# Result of a dup/dup2 system call

- Both Process table entries point to the same file table entry



# File Control using fcntl()

- Control (change/get) the properties of a file that is already open

```
int fcntl(int fd, int cmd, /* int arg
    */...);
```

– Cmd is

- F\_DUPFD: duplicate existing file descriptor
- F\_GETFD/F\_SETFD: Get/Set file descriptor flags
- F\_GETFL/F\_SETFL: Get/Set file status flags
- F\_GETOWN/F\_SETOWN: File Async I/O ownership
- F\_GETLK/F\_SETLK: Get/Set record locks

# Device Operations using ioctl()

- Handle devices based on the flags set

```
int ioctl(int fd, int req, ...);
```

- req signifies the operation performed on the device/file associated with fd
- The ioctl() is like a placeholder for file/device specific operations that are not done by other system calls

# File information using `*stat()`

- Give information about a file

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

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

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

- Return information about the file denoted by path (`stat/lstat`) or fd (`fstat`) into buf
  - In case of `lstat`, return information about the softlink than of the original file.
- The `stat` structure defined in `<sys/stat.h>`

# File types

- Common file types
  - Regular
  - Directory
  - Character Special
  - Block Special
  - FIFO/named pipes
  - Socket
  - Symbolic link

# File Access Permissions

- User, Group, Other level permissions for
  - Read, write, execute
- File/directory access checks when a file is
  - Read
  - Modified
  - Deleted
  - Executed

# Access checks using access()

- Check access permissions for the current user

```
int access(const char *path, int mode)
```

– Check access for file indicated by path, where mode is

- R\_OK: test for reading
- W\_OK: test for writing
- X\_OK: test for execution
- F\_OK: test for existence

# File mode creation mask using umask()

- Set the file mode creation mask to the given value, and return the existing mask

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

- cmask is either 0 or OR-ed value of
  - S\_IRUSR, S\_IWUSR, S\_IXUSR
  - S\_IRGRP, S\_IWGRP, S\_IXGRP
  - S\_IROTH, S\_IWOTH, S\_IXOTH

# File mode changes using chmod/fchmod

- Change the file mode

```
int chmod(const char *path, mode_t mode);  
int fchmod(int fd, mode_t mode)
```

- Set the file mode to mode, using
  - S\_IRUSR, S\_IWUSR, S\_IXUSR
  - S\_IRGRP, S\_IWGRP, S\_IXGRP
  - S\_IROTH, S\_IWOTH, S\_IXOTH

# Change file ownership

- Change ownership

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

```
int fchown(int fd, uid_t owner, gid_t  
          group);
```

```
int lchown(const char *path, uid_t owner,  
          gid_t group);
```

- Change the owner to owner, group to group.

# Filesystem basics

- i-nodes
  - Contain information about where the data for a file exists
- Directory
  - Contains information about each file and a pointer to corresponding inode
- Multiple filenames can point to the same i-node, effectively leading to the same file.
  - The original entry for such file is a normal file entry
  - The other entries are links (hard links)

# Creating/Deleting files/links

- Create a hardlink

```
int link(const char *existingfile, const
char *newfile);
```

- Remove a file (only when it is not open)

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

- Remove a file (now)

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

- Rename a file (only when it is not open)

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

# Symbolic links

- Symbolic links are references to other files
  - They can exist even if the target doesn't exist
  - They can work across filesystems

- Creating symbolic links

```
int symlink(const char *actualfile, const
            char *symfile);
```

- Reading the name pointed to by a symbolic link

```
int readlink(const char *file, char *buf,
             int bufsize);
```

# Directory Operations

- **Creating Directories**

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

- **Removing Directories**

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

- **Finding Current Directory**

```
char *getcwd(char *buf, size_t size);
```

- **Changing Directories**

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

```
int fchdir(int fd);
```

# Standard I/O library

- Why library functions that mimic system call functionality?
  - Customize for efficiency
    - Example: buffer contents till you need to write to it
  - Custom implementations that work across OS types etc.

# Standard I/O library

- Why library functions that mimic system call functionality?
  - Customize for efficiency
    - Example: buffer contents till you need to write to it
  - Custom implementations that work across OS types etc.

# Unix Standard I/O library

- FILE object
  - The FILE object is associated with I/O streams
  - A pointer to FILE object is used in most I/O operations
- Standard I/O
  - The file descriptors are
    - STDIN\_FILENO, STDOUT\_FILENO, STDERR\_FILENO
  - The FILE pointers are
    - stdin, stdout, stderr

# Buffered I/O

- Buffering is used to minimize the number of calls to kernel
- Buffering offered by standard I/O
  - Fully buffered: Buffer contents until buffer becomes full.
  - Line buffered: Buffer contents until a newline appears
  - Unbuffered: Directly send data to the file

# Buffering Characteristics

- ANSI-C:
  - Standard input/output are fully buffered if they do not refer to an interactive device
  - Standard error is never fully buffered
- SVR-4/4.3BSD:
  - Standard error is always unbuffered
  - Other streams are line buffered if they are associated with an interactive device. Fully buffered otherwise.

# Changing the buffering behaviour

- Set the characteristics of buffer

```
void setbuf(FILE *fp, char *buf);
```

```
int setvbuf(FILE *fp, char *buf, int mode,  
            size_t size);
```

– setbuf: buf is of size BUFSIZ

– setvbuf: mode is one of

- \_IOFBUF
- \_IOLBUF
- \_IONBUF

# Flushing a buffer

- Flush the buffer associated with the file

```
int fflush(FILE *fp);
```

- Flush the contents of buffer associated with fp

# Opening a stream

- Open a standard i/o stream

```
FILE *fopen(const char *file, const char
            *type);
```

```
FILE *freopen(const char *file, const char
              *type, FILE *fp);
```

```
FILE *fdopen(int fd, const char *type);
```

– Type is one of

- “r” or “rb” or “r+” or “r+b” or “rb+”
- “w” or “wb” or “w+” or “w+b” or “wb+”
- “a” or “ab” or “a+” or “a+b” or “ab+”

# Closing a stream

- Close a standard i/o stream

```
int fclose(FILE *fp);
```

# Character I/O

- Character at-a-time I/O

```
int getc(FILE *fp);  
int fgetc(FILE *fp);  
int getchar(); /* same as getc(stdin) */  
int putc(int c, FILE *fp);  
int fputc(int c, FILE *fp);  
int putchar(int c); /* same as putc(c,  
    stdout) */  
int ungetc(int c, FILE *fp);
```

# Stream status checks

- Status checks

```
int ferror(FILE *fp);
```

```
int feof(FILE *fp);
```

- Check if the stream has error status or at eof.

```
void clearerr(FILE *fp);
```

- Clear the error status of the stream

# Line I/O

- Line-at-a-time I/O

```
char *fgets(char *buf, int n, FILE *fp)
```

```
char *gets(char *buf);
```

```
int fputs(const char *buf, FILE *fp);
```

```
int puts(const char *buf);
```

- Clear the error status of the stream

# Formatted I/O

- Specify the format of the string and the arguments

```
int printf(const char *format, ...);
```

```
int fprintf(FILE *fp, const char  
*format, ...);
```

```
int sprintf(char *buf, const char  
*format, ...);
```

```
int scanf(const char *format, ...);
```

```
int fscanf(FILE *fp, const char  
*format, ...);
```

```
int sscanf(const char *buf, const char  
*format, ...);
```

# Formatted I/O, continued

- Specify the format of the string and the arguments in a `va_list`

```
int printf(const char *format, va_list  
    arg);
```

```
int fprintf(FILE *fp, const char *format,  
    va_list arg);
```

```
int sprintf(char *buf, const char *format,  
    va_list arg);
```

# Binary I/O

- Write/read binary data

```
size_t fread(void *ptr, size_t size, size_t  
nobj, FILE *fp);
```

```
size_t fwrite(const void *ptr, size_t size,  
size_t nobj, FILE *fp);
```

# Position in a stream

- Check/set the position in a stream

```
long ftell(FILE *fp);
```

```
int fseek(FILE *fp, long offset, int  
whence);
```

```
void rewind(FILE *fp);
```

```
int fgetpos(FILE *fp, fpos_t *pos);
```

```
int fsetpos(FILE *fp, const fpos_t *pos);
```