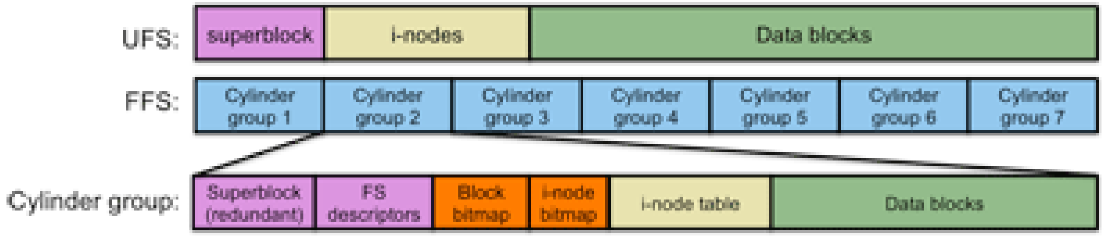
Systems Software/Programming

File IO

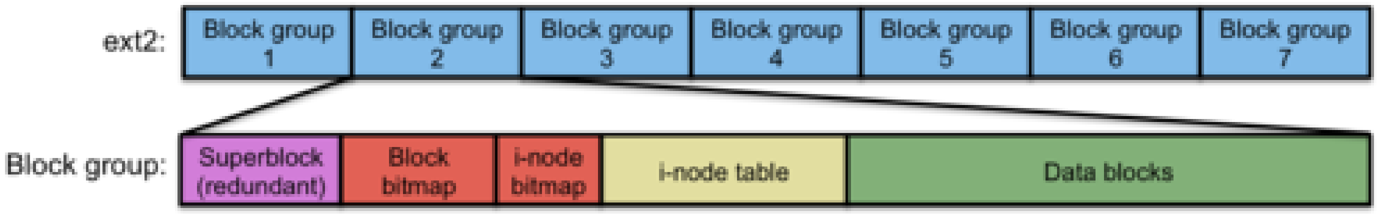
Files and Directories in Unix based systems

Todays File Systems Extended from UFS (Unix File System)

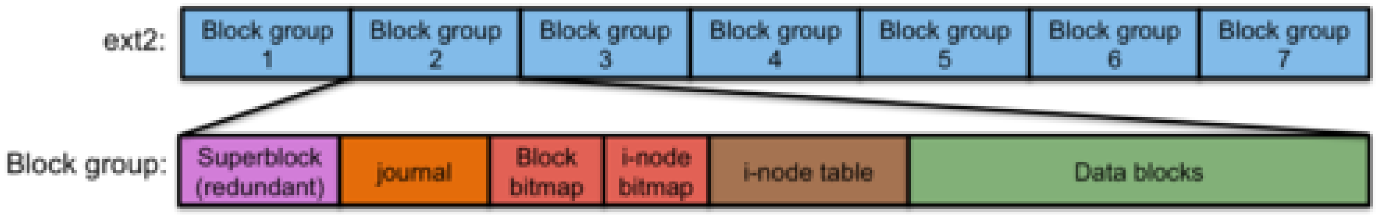
- BSD Fast File System



- Linux ext2

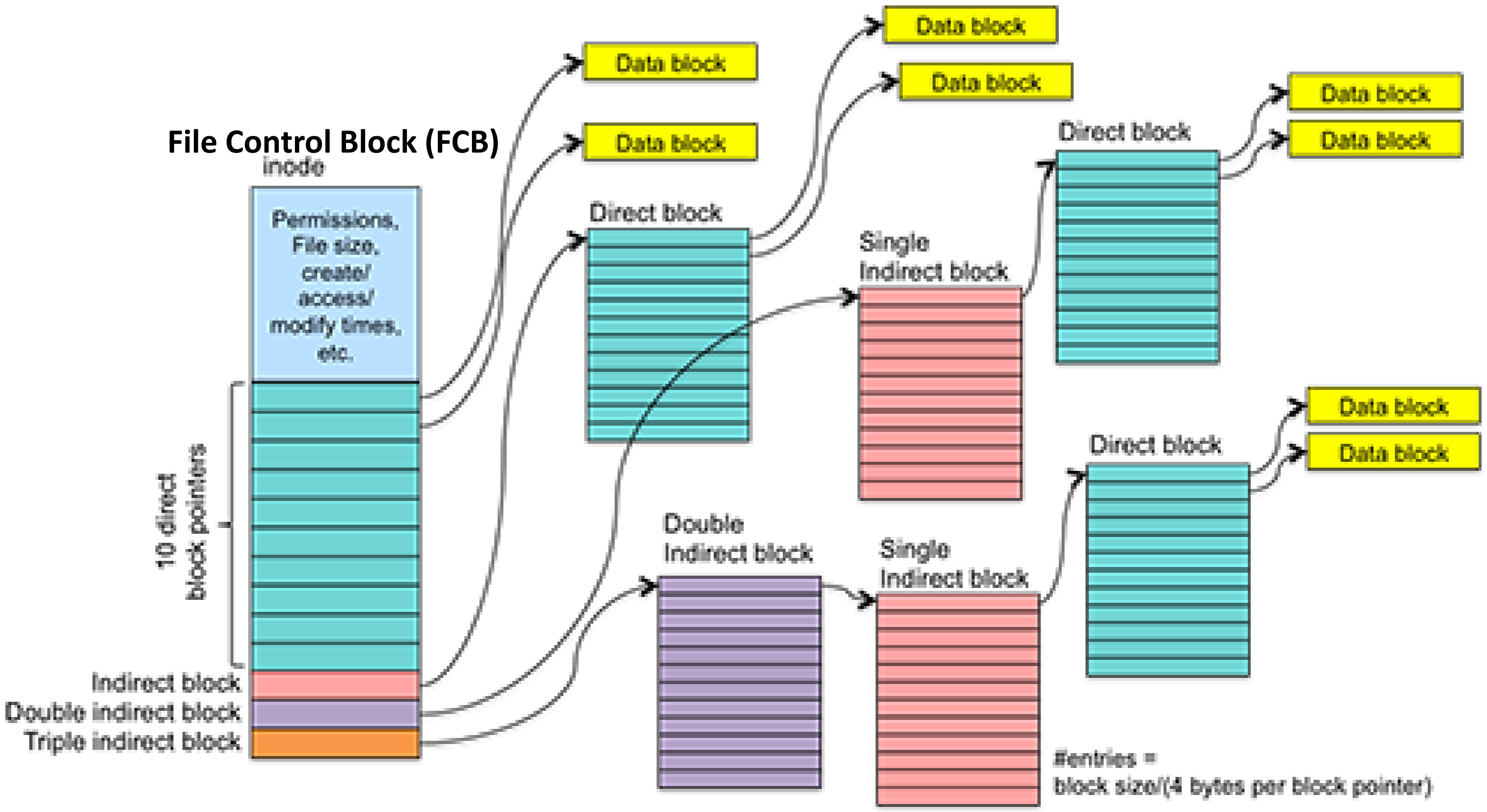


- Linux ext3



- Linux ext4 – Structure is same as ext3 with some addition (only 2 are listed here)
  - Large file system (disks upto 1 exabytes vs 32 TB in ext3 and file size upto 16TB vs 2TB in ext3)
  - Larger Directories 64000 entries vs 32000 in ext3

Unix File System (UFS) inode layout in disks



UFS vs EXT Superblock

- UFS superblock (Volumn Control Block) contains
  - Size of file system
  - Number of free blocks
  - list of free blocks (including a pointer to free block lists)
  - index of the next free block in the free block list
  - Size of the inode list
  - Number of free inodes in the file system
  - Index of the next free inode in the free inode list
  - Modified flag (clean/dirty)
- FFS/EXT superblock (Volumn Control Block) contains
  - Blocks in the file system
  - No of free blocks in the file system
  - Inodes per block group
  - Blocks per block group
  - No of times the file system was mounted since last fsck.
  - Mount time
  - UUID of the file system
  - Write time
  - File System State (i.e.: was it cleanly unmounted, errors detected etc.)
  - The file system type etc.(i.e.: whether its ext2,3 or 4).
  - The operating system in which the file system was formatted

inode File Control Block (FCB)

- UFS inode contains (aka File Control Block FCB)
  - file owner, group owner
  - file type (regular file, directory, character device, block device, FIFO)
  - access permissions
  - times: create, last access, and last modification
  - number of links to the file, which is the number of file names in the directory that refer to this inode
  - file size or major/minor numbers if the file is a device
  - file data location: ten direct blocks, one indirect block, one double indirect block, and one trible indirect block

See the inode information

For Currrent Directory

faculty@faculty-OptiPlex-3040:~$ ls -lia

- total 134204
- 9699330 drwxr-xr-x 57 faculty faculty 4096 Dec 21 10:59 .
- 9699329 drwxr-xr-x 3 root root 4096 Feb 24 2017 ..
- 9699466 -rw-rw-r-- 1 faculty faculty 94319943 Mar 21 2017 2.4.13
- 12323363 drwxrwxr-x 3 faculty faculty 4096 Aug 29 13:29 ApacheSpark
- 10094452 drwxrwxr-x 3 faculty faculty 4096 Jun 28 12:03 BackupUSB
- 9699526 -rw------- 1 faculty faculty 49233 Dec 21 13:23 .bash_history
- 9699331 -rw-r--r-- 1 faculty faculty 220 Feb 24 2017 .bash_logout
- 9705098 -rw-r--r-- 1 faculty faculty 4330 Nov 24 14:17 .bashrc
- 9704737 -rw-r--r-- 1 faculty faculty 4198 Aug 23 13:30 .bashrc~

Recursively for current and all subdirectories

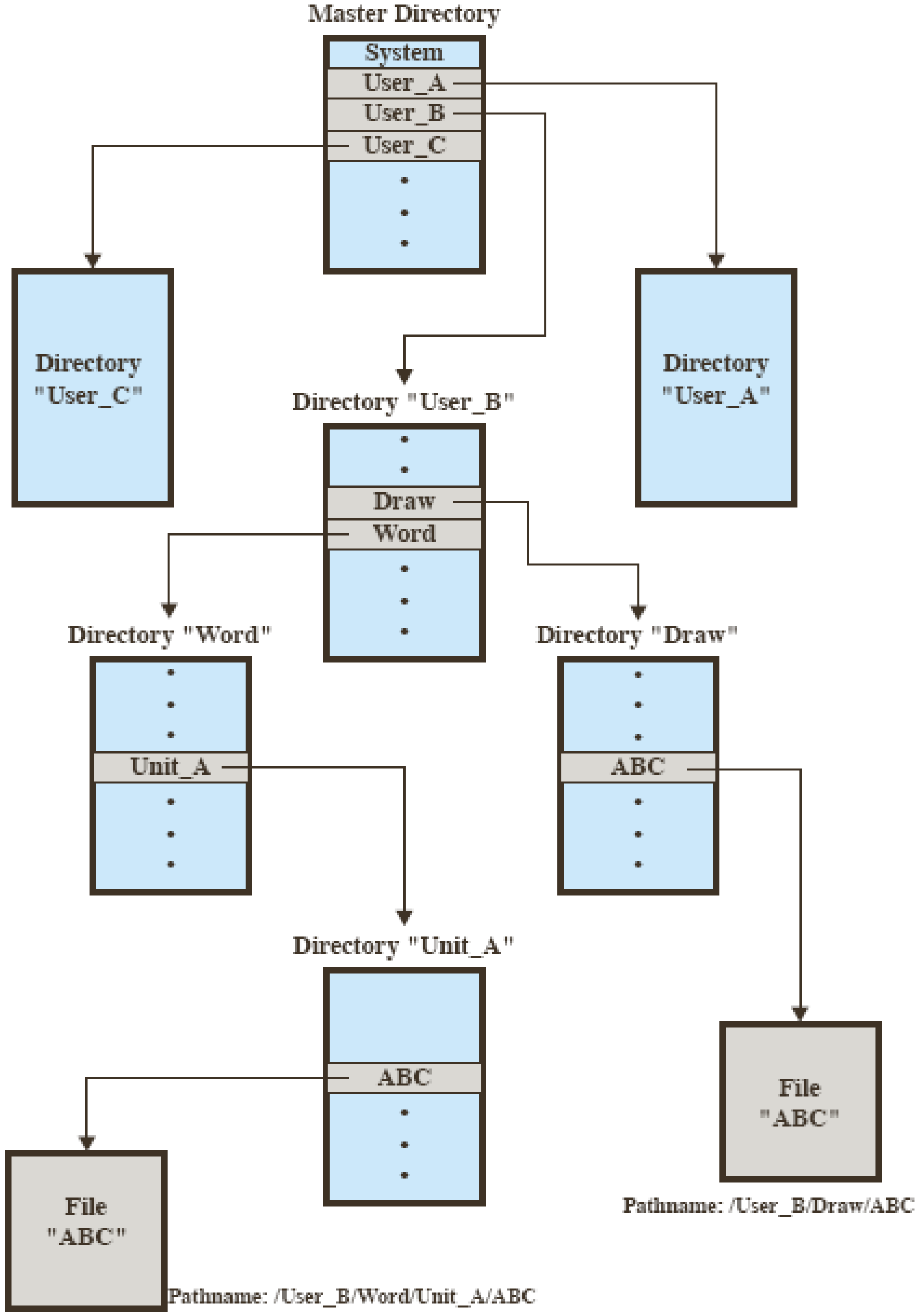
faculty@faculty-OptiPlex-3040:~$ ls -iaR virtualenv/ | more

- virtualenv/:
- 9704509 .
- 9699330 ..
- 11540615 architectureanalysis
- 9704475 bashrcn
- 9704543 bashrcn~
- 9705036 hosts
- 9704703 hosts~
- 9699741 ml
- 9704714 mpi_pi.c
- 9704716 mpi_pi.out

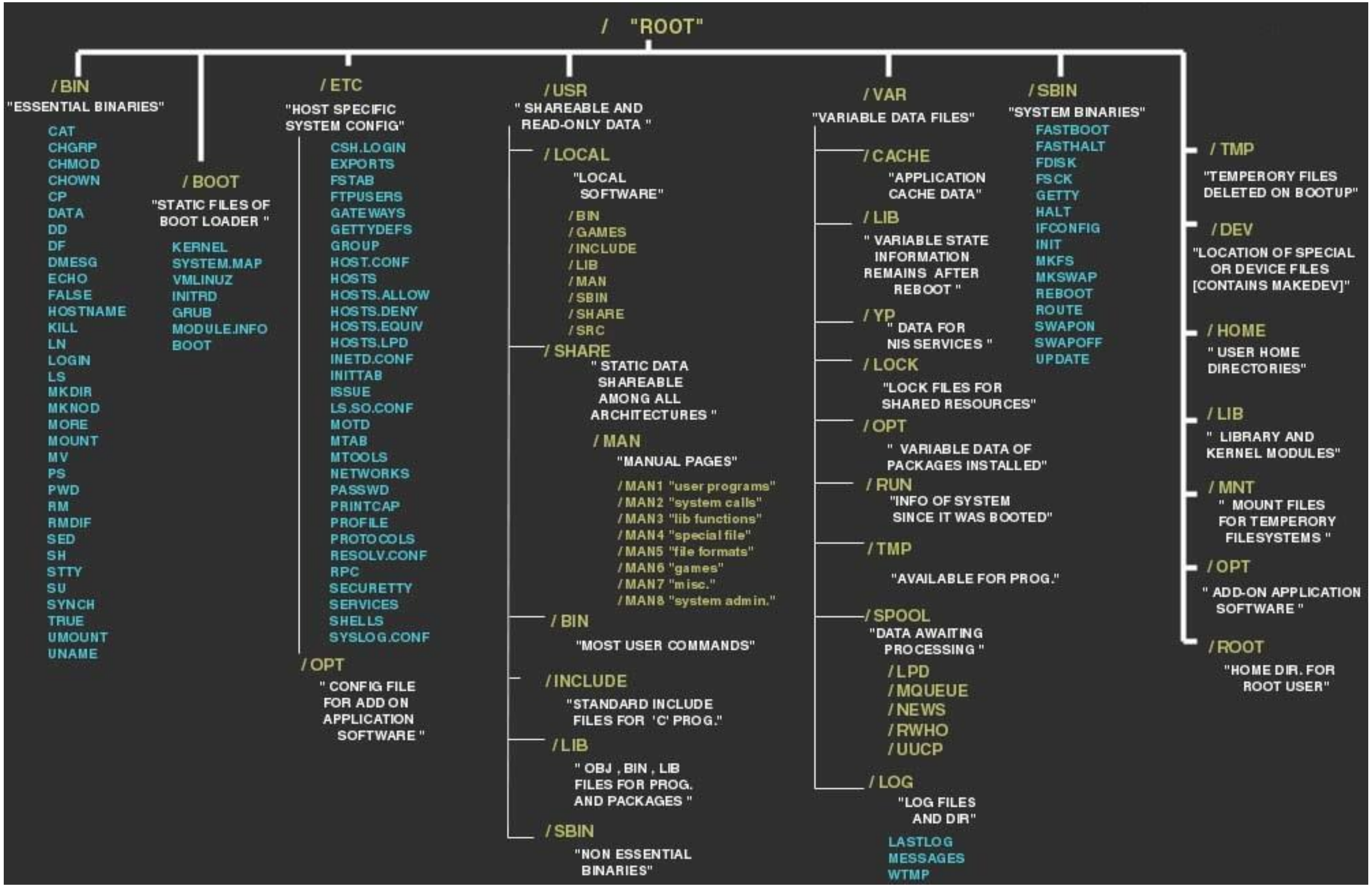
- virtualenv/architectureanalysis:
- 11540615 .
- 9704509 ..
- 11407369 archanalyze_decisiontree.ipynb
- 11540737 archanalyze_linearregression.ipynb
- 11541142 archanalyze_mlp-Copy1.ipynb
- 11540670 archanalyze_mlp.ipynb
- 9312887 archanalyze_mlp.py
- 11540730 archanalyze_partialleastsquare.ipynb

Directories

- Directories are special files that keep track of other files
  - the collection of files is systematically organized
  - first, disks are split into partitions that create logical volumes (can be thought of as “virtual disks”)
  - second, each partition contains information about the files within
  - this information is kept in entries in a **device directory** (or volume table of contents)
  - the directory is a symbol table that translates file names into their entries in the directory
    - it has a logical structure
    - it has an implementation structure (linked list, table, etc.)

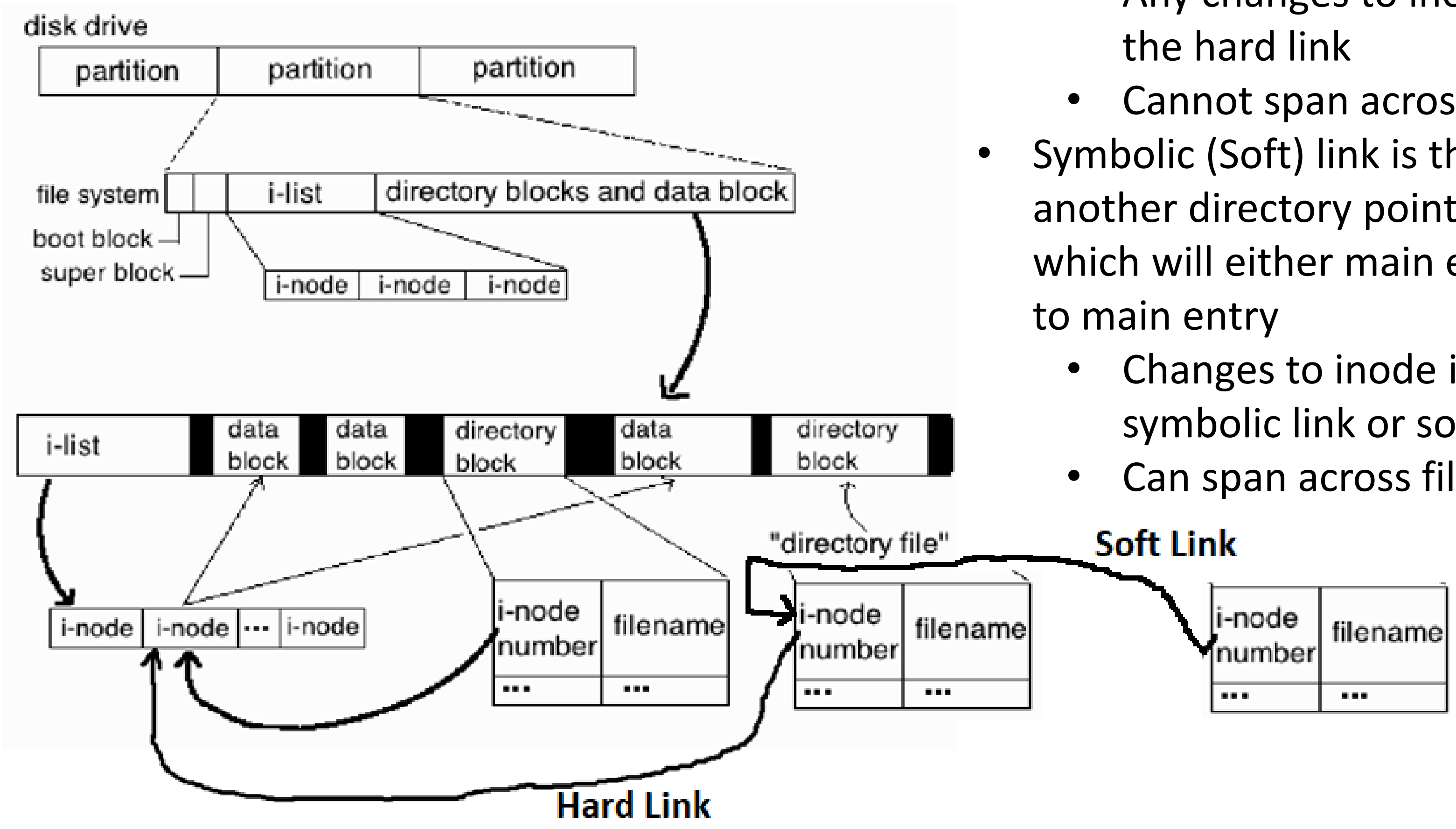


Linux File System Hierarchy



Directory entries pointing to file with link - Two directory entries refer to the same file but with different names

- Hard link is the directory entry of a file or another directory pointing to inode directly
  - Any changes to inode of original file does not affect the hard link
  - Cannot span across file systems
- Symbolic (Soft) link is the directory entry of a file or another directory pointing to another directory entry which will either main entry or will have hard or soft link to main entry
  - Changes to inode in original file will affect the symbolic link or soft link
  - Can span across file system



Create Symbolic Links (ln command)

Hard Link

ln mount_HostFiles.sh mnt

$ ls -lia m*

138633 -rwxrwxrwx 2 ubuntu18 ubuntu18 92 Jan 4 14:34 mnt

138633 -rwxrwxrwx 2 ubuntu18 ubuntu18 92 Jan 4 14:34 mount_HostFiles.sh

inode is same for hard link as original file

Soft Link

ln -s mount_HostFiles.sh mnt_soft

$ ls -lia m*

133913 lrwxrwxrwx 1 ubuntu18 ubuntu18 18 Jan 28 09:37 mnt_soft -> mount_HostFiles.sh

138633 -rwxrwxrwx 2 ubuntu18 ubuntu18 92 Jan 4 14:34 mount_HostFiles.sh

inode is different for soft link then original file

File Properties

```
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
}
```

- Stat Functions

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

- Returns information about a named file

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

- Returns information about already opened file

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

- Returns information about symbolic link, not the referenced file

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

st_ino and st_nlinks

- st_nlinks – provides number of hardlinks
- Create hard link for a file and run this code by passing that filename or hardlink
- File_IO\stat_links.c

st_mode bits

- File Types: S_IFREG – regular file, S_IFDIR – directory, S_IFLINK – symbolic link
- Macro to Check for File Types: S_ISREG(mode) – check if mode value indicates regular file, S_ISDIR(mode) – check if mode indicates directory, S_ISLNK(mode) – check if mode indicates symbolic link
- Permission: Owner

S_IRWXU - read, write, execute/search by owner (bitwise OR of S_IRUSR, S_IWUSR, S_IXUSR)

S_IRUSR- read permission, owner

S_IWUSR - write permission, owner

S_IXUSR - execute/search permission, owner

Similarly GRP/G instead of USR/U indicates permission for Group and OTH/O indicates permission for Other Users

[File_IO\stat_example.c]

Useful Library Functions – The same are also available as commands for shell

#include <unistd.h>

int access(const char *pathname, int mode) // get accessibility i.e. file permissions

int chown(const char *pathname, uid_t owner, gid_t group) // change owner of file to provided values in owner an group parameters

int truncate(const char *pathname, off_t length) // truncate the file size to length parameter

int link(const char *existingpath, const char *newpath) // creates hard link newpath from the existing file at existingpath

int ulink(const char *pathname) // remove hard link

int symlink(const char *actualpath, const char *sympath) // create symbolic (soft) link

int readlink(const char *restrict pathname, char *restrict buf, size_t bufsize) // read the content of symbolic (soft) link, performs open(), read() and close() function all together

int rmdir(const char *pathname) // removes directory

Useful Library Functions – The same are also available as commands for shell

#include <stdio.h>

int remove(const char *pathname) // delete file or directory or unlink symbolic link

int rename(const char *oldname, const char *newname) // rename file or directory

#include <sys/stat.h>

int chmod(const char *pathname, mod_t mode) // change file access permission

int mkdir(const char *pathname, mode_t mode) // creates new directory

dirent.h – format of directory entries

- The internal format of directories is unspecified.
- The <dirent.h> header defines the following data type through typedef:
- DIR : A type representing a directory stream.
- It also defines the structure dirent which includes the following members:
  - ino_t d_ino file inode (aka serial) number
  - char d_name[] name of entry
  - The type ino_t is defined as described in <sys/types.h>.
  - The character array d_name is of unspecified size, but the number of bytes preceding the terminating null byte will not exceed {NAME_MAX}.

Directory Functions

#include <dirent.h>

DIR *opendir(const char *pathname); // open directory stream and return pointer to DIR stream object (DIR is similar to FILE stream object)

struct dirent *readdir(DIR *dp); // read directory entries from DIR stream into pointer to (array of) structure dirent which has inode numbers and filenames

void rewinddir(DIR *dp); // rewind DIR stream to beginning of directory

int closedir(DIR *dp); // close DIR stream

long telldir(DIR *dp); // return the current position of DIR stream

void seekdir(DIR *dp, long loc); //seek to location/position mentioned by loc

Implementation of pwd (present working directory) using Directory Functions

Implement present working directory functionality of shell command that displays the path of the current directory.

File_IO\mypwd.c

Buffered vs Unbuffered IO

- Unbuffered I/O: each read write invokes a system call in the kernel.
  - read, write, open, close, lseek
  - Data Unit: raw byte
- Buffered I/O: data is read/written in optimal-sized chunks from/to disk --> streams
  - standard I/O library written by Dennis Ritchie
  - Data Unit: C data type

Unbuffered IO

Open a File

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h> or <unistd.h>

int open(const char *pathname, int flags);

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

- Parameters
  - pathname : name of the file with complete path
  - flags:
    - O_RDONLY : read-only access
    - O_WRONLY : write-only access
    - O_RDWR : read-write access
    - O_CREAT: if file doesn't exists then create it
    - O_APPEND : in write mode, don't overwrite but append the new content
    - O_TRUNC : in write mode, truncate the file before writing new content
  - mode:
    - 0600 (i.e. –rwx------) : read-write access for current user, no access for group users or other users
    - 0644 (i.e. –rwxr—r--) : read-write access for current user, read-only access for group users and other users

Open a File

- open() returns an integer:
  - -1 means error i.e. file could not be opened
  - >= 0 : this is a “file descriptor” of a open file. Save it in a variable, you will need to pass it to all subsequent file related functions such as read, write etc
- You don’t need to specify file mode unless you will be creating a new file
  - fd = open(“this_file_already_exists”, O_RDONLY);
- You can combine multiple flags together
  - fd = open(“foo”, O_RDWR | O_CREAT, 644);

Operating System Tables

Process File Descriptor Table

	fd Flags	File Pointer
0		
1		
2		
3		
⋮		
P		

System-Wide Open File Table

	File Offset	Status Flags & Access Mode	Reference Count	i-Node Pointer
0				
1				
2				
3				
4				
5				
6				
7				
8				
⋮				
N				

System-Wide i-Node Table

	File Type	File Locks	File Properties
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
⋮			
M			

Process File Descriptor Table

	fd Flags	File Pointer
0		
1		
2		
⋮		
Q		

- If file exists
  - Create an entry in System-Wide Open File Table
  - Create entry in Process File Descriptor Table pointing to System-Wide Table
- If file needs to be created
  - Disk FCB (i-node) is created first and then loaded it in System-Wide i-node Table
  - Then same two steps from above

dup and dup2

#include <unistd.h>

int dup(int oldfd); → creates a copy of the file descriptor oldfd, using the lowest-numbered unused file descriptor for the new descriptor

int dup2(int oldfd, int newfd); → create a new file descriptor newfd pointing to the same physical file as oldfd

Returns:

New file descriptor on success

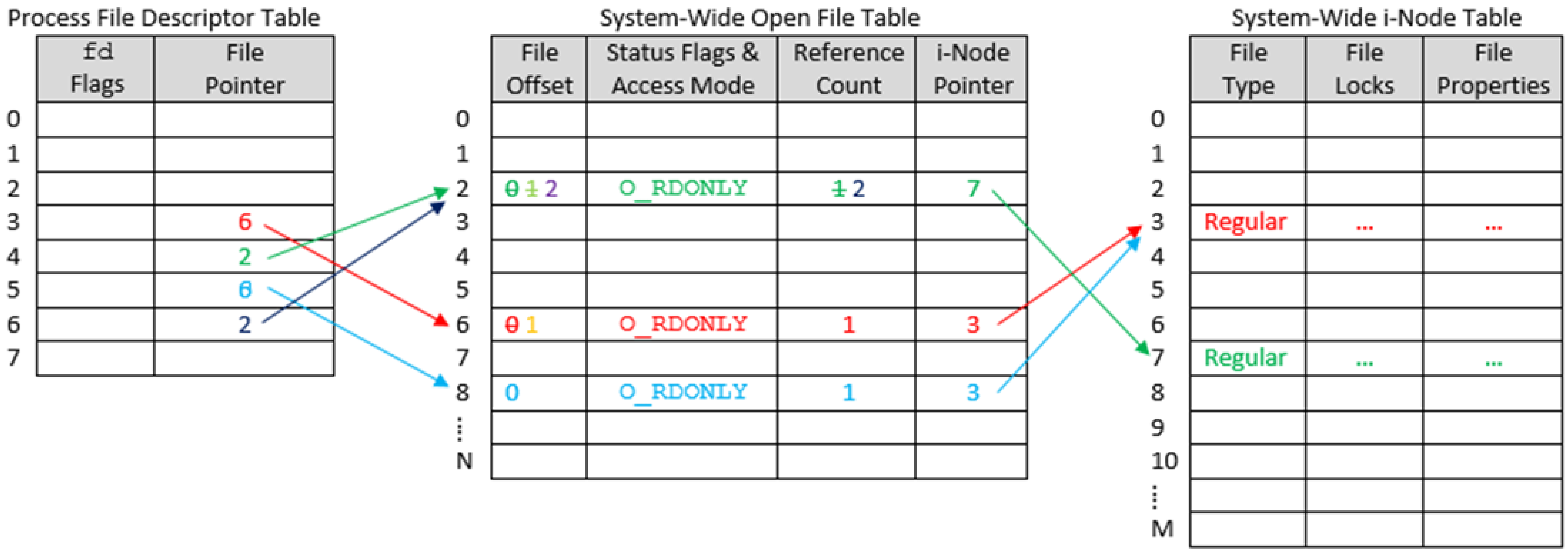
-1 on error with **errono** variable set to check for exact error

Read From a File

```
#include <sys/types.h>
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```
→ difference betn size_t & ssize_t?

- Parameters:
  - fd : file descriptor of the file which you want to read from
  - buf : buffer where the file content will be stored after reading
  - count : number of bytes to read
- read() returns an integer:
  - -1 means error reading the file
  - >= 0 : number of bytes that were actually read from the file. If return value is less then the value in count (i.e. number of bytes to be read) then it is inferred that End of File has reached

Difference between open and dup




```
fdA1 = open("fileA.txt", O_RDONLY);
read(fdA1, &c, 1);
fdB  = open("fileB.txt", O_RDONLY);
read(fdB, &c, 1);
fdA2 = open("fileA.txt", O_RDONLY);
fdBdup = dup(fdB);
read(fdBdup, &c, 1);
```

Write Into a File

```
#include <sys/types.h>
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

- Parameters:
  - fd : file descriptor of the file which you want to write to
  - buf : buffer from where the content will be written to the file
  - count : number of bytes to write
- write() returns an integer:
  - -1 means error writing the file
  - >= 0 : number of bytes were actually written into the file which should be the same as value in count. If return value is less then the value in count then you may have encountered error like not sufficient disk space etc.

Close a File

<unistd.h> is printed after #include:

```
#include <unistd.h>
int close(int fd);
```

- Parameters:
  - fd : file descriptor of the file which you want to close
- close() returns an integer:
  - -1 means error closing the file
  - 0 : OK (i.e. file successfully closed)
- Don’t forget to close the file once you have finished using the file otherwise you leave orphan file descriptors in the system. (Normally OS will check all the open files at the time when program execution ends and will closes them but we must close them in our program.

Difference between open and dup

Process File Descriptor Table

	fd Flags	File Pointer
0		
1		
2		
3		6
4		2
5		8
6		2
7		

System-Wide Open File Table

	File Offset	Status Flags & Access Mode	Reference Count	i-Node Pointer
0				
1				
2	~~0~~ ~~1~~ 2	O_RDONLY	~~1~~ 2	7
3				
4				
5				
6	~~0~~ 1	O_RDONLY	1	3
7				
8	0	O_RDONLY	1	3
⋮				
N				

System-Wide i-Node Table

	File Type	File Locks	File Properties
0			
1			
2			
3	Regular	...	...
4			
5			
6			
7	Regular	...	...
8			
9			
10			
⋮			
M			

```
fdA1 = open("fileA.txt", O_RDONLY);
read(fdA1, &c, 1);
fdB  = open("fileB.txt", O_RDONLY);
read(fdB, &c, 1);
fdA2 = open("fileA.txt", O_RDONLY);
fdBdup = dup(fdB);
read(fdBdup, &c, 1);
```

What happens when we do close(fdA1)?

How about close(fdB)?

Special Files

- When any program start executing, 3 file descriptors are created automatically
  - You can use them as you like
  - You are not obliged to close them after you finish using them
- These file descriptors are:
  - 0: it represents standard input (generally keyboard) for your program. When user types input on keyboard it is read using file descriptor 0. It is read-only file descriptor which can’t be written
  - 1: it represents standard output (generally display screen) for your program. When any message is written file descriptor 1, it is displayed on the screen
  - 2: it represents standard error (generally display screen) for your program. You can use this to display any error messages to screen.

Seeking a File

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

Returns: new file offset if OK, -1 on error

whence: where to start the offset i.e. SEEK_SET, SEEK_CUR, SEEK_END

SEEK_SET – file offset set to offset

SEEK_CUR – file offset set to Current location + offset

SEEK_END – file offset set to file size + offset (i.e. increasing the file size)

Examples

- Example with lseek : [File_IO\FileIO_Example1.c]
- Example with multiple File Descriptor to the same file and using STDOUT file descriptor with write() system call : [File_IO\FileIO_Example2.c]

Buffered IO

Standard I/O Library

- Difference from Unbuffered File I/O
  - File Pointers (FILE *) vs File Descriptors (int)
  - fopen vs open
    - When file is opened or created, a stream is associated with the file.
    - FILE object
      - File descriptor, buffer size, # of remaining chars, an error flag
  - stdin, stdout, stderr defined in <stdio.h>
    - STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO (unbuffered file descriptors are defined in <unistd.h>) as IEEE POSIX (Portable Operating System Interface) standard

Buffering

- Goal
  - Use minimum number of read and write calls
- Types
  - Fully Buffered
    - Actual I/O occurs when the buffer is filled up
    - A buffer is automatically allocated when first I/O operation is performed on a stream
    - flush: standard I/O library vs terminal driver
  - Line Buffered
    - Perform I/O when a newline character is encountered – usually for terminals
    - Caveat
      - Filling of buffer could trigger I/O
      - Flushing all line buffered output if input requested
  - Unbuffered
    - Expect to output ASAP e.g. when write() is called
    - Or when using stdout or stderr

Buffering

By default, buffering is done automatically when file is opened but in case you need a specific buffer implementation you can use this functions

#include <stdio.h>

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

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

- Full/line buffering if buf is NOT NULL (BUFSIZ)
  - Terminals
- Mode _IOFBF, _IOLBF, _IONBF (<stdio.h>)
  - Optional size → st_blksize (stat())
- #define BUFSIZ 1024 (<stdio.h>)
- They must be called before any operation is performed on the stream

Buffering

- ANSI C requirements
  - Fully buffered for stdin and stout unless interactive devices are referred to.
    - 4.3+BSD – line buffered
  - Stderr in never fully buffered

```
#include <stdio.h>
int fflush(FILE *fp);
```

  - All output stream are flushed if fp == NULL

Opening a Stream

- #include <stdio.h>
- FILE *fopen(const char *pathname, const char *type);

- opens a specified file
- types:
  - r : open for reading
  - w : create for writing or truncate to 0
  - a : open or create for writing at the end of file
  - r+ : open for reading and writing
  - w+: create for reading and writing or truncate to 0
  - a+ :open or create for reading and writing at the end of file

  - use b to differentiate text vs binary , e.g. rb, wb ..etc

fopen() vs open() flags

fopen() flags	open() flags
r	O_RDONLY
w	O_WRONLY | O_CREAT | O_TRUNC
a	O_WRONLY | O_CREAT | O_APPEND
r+	O_RDWR
w+	O_RDWR | O_CREAT | O_TRUNC
a+	O_RDWR | O_CREAT | O_APPEND

Restrictions

Type	r	w	a	r+	w+	a+
File exists?	Y			Y		
Truncate		Y			Y	
R	Y			Y	Y	Y
W		Y	Y	Y	Y	Y
W only at end			Y			Y

- When a file is opened for reading and writing:
  - Output cannot be directly followed by input without an intervening *fseek, fsetpos, or rewind*
  - Input cannot be directly followed by output without an intervening *fseek, fsetpos, or rewind*

Setting File Position

- long ftell(FILE * stream)
  - obtains the current value of the file position indicator for the stream pointed to by stream.
- int fseek (FILE * stream, long int offset, int whence);
  - stream: Pointer to a FILE object that identifies the stream.
  - offset : Binary files- Number of bytes to offset from *whence*. Text files- Either zero, or a value returned by ftell().
  - whence: SEEK_SET - Beginning of file, SEEK_CUR - Current position of the file pointer, SEEK_END - End of file *
- int rewind(FILE * stream);
  - Rewind function sets the file position indicator for the stream pointed to by stream to the beginning of the file. It is equivalent to

  (void)fseek(stream, 0L, SEEK_SET)

Setting File Position

- fgetpos() and fsetpost() alternate interfaces equivalent to ftell() and fseek() with whence set to SEEK_SET
- int fgetpos(FILE *stream, fpos_t *pos);
- int fsetpos (FILE * stream, const fpos_t * pos);
  - Stream: pointer to a FILE object that identifies the stream
  - pos: Pointer to a fpos_t object containing a position previously obtained with fgetpos.

Closing a Stream

#include <stdio.h>

int fclose(FILE *fp);

- Flush buffered output
- Discard buffered input
- All I/O streams are closed after the process exists

- setbuf or setvbuf to change the buffering of a file before any operation on the stream

Reading and Writing from/to Streams

- Unformatted I/O
  - Character-at-a-time I/O e.g. getc()
    - Buffering handled by standard I/O library
  - Line-at-a-time I/O e.g. fgets()
    - Buffer limit might need to be specified
  - Direct I/O e.g. fread()
    - Read/write a number of objects of a specified size
    - An ANSI C term e.g. = object-at-a-time I/O

Reading a Char

#include <stdio.h>

int getc(FILE *fp); - Can be used as a macro or a function

int fgetc(FILE * fp); - Can be used only as a function

int getchar(void);

- Getchar() == getc(stdin)
- unsigned char converted to int in returning

Error/EOF Check

#include <stdio.h>

int ferror(FILE *fp); - test error indicator

int feof(FILE *fp); - test if end of file

void clearerr(FILE *fp);

int ungetc(int c, FILE *fp); - puts the char back in the stream which was read using getc

- An error flag and EOF flag for each FILE
- No pushing back of EOF (i.e. -1)
  - No need to be same char read!

Writing a char

```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

- putchar(c) == putc(c, stdout)
- Differences between putc and fputc
  - putc() can be implemented as a macro hence can’t be used in function pointer

Example

getchar(), putchar() example : File_IO\FileIO_Example3.c

Copy File Example: File_IO\copy_file.c

./copy_file.out abc.txt def.txt 2>&1

2>&1 can be used to redirect both stderr and stdout to stdout

Line-at-a-Time I/O

#include <stdio.h>

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

- Include ‘\n’ and be terminated by null
- Could return a partial line if the line is too long

char *gets(char *buf);

- Read from stdin
- No buffer size is specified → overflow
- *buf does not include ‘\n’ and is terminated by null

Line-at-a-Time I/O

#include <stdio.h>

char *fputs(const char *str, FILE *fp);

- Include ‘\n’ and be terminated by null
- No need for line-at-a-time output

char *puts(const char *str);

- *str does not include ‘\n’ and is terminated by null
- puts then writes ‘\n’ to stdout

Example 2

fgets() fputs() example: File_IO\FileIO_Example4.c

Copy file example: File_IO\copy_file.c

Standard I/O Efficiency

- Measure time spent in user mode and kernel mode using : time <program>
- Copy file File_IO\IOefficiency.txt of size 2,100,000 bytes (@2MB) to File_IO\IOefficiency1.txt by redirecting stdin and stdout: File_IO\IOefficiency.c

time ./IOefficiency.out <IOefficiency.txt >IOefficiency1.txt

Functions	Total (Real)	User Time	Kernel Time
fgets(), fputs() (5MB buffer)	0m0.144s	0m0.006s	0m0.057s
fgetc(), fputc()	0m0.190s	0m0.008s	0m0.100s
read(), write()	6m27.971s	0m0.687s	2m35.339s

Reading fixed sized items

#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);

- ptr – pointer to buffer to which data will be read from stream
- size – size of individual item
- nitems – number of items
- stream – FILE stream object

What is the difference in the below calls?

elements_read = fread(buf, sizeof(buf), 1, fp);

bytes_read = fread(buf, 1, sizeof(buf), fp);

Writing fixed size items

#include <stdio.h>

size_t fwrite(const void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);

- ptr – pointer to buffer from which data has to be written to stream
- size – size of individual item
- nitems – number of items
- stream – FILE stream object

Reading and Writing C Data types

- int fscanf(FILE *stream, const char *format, ...);
- int fprintf(FILE *stream, const char *format, ...);
- Same as scanf and printf except operations are performed on a given file stream