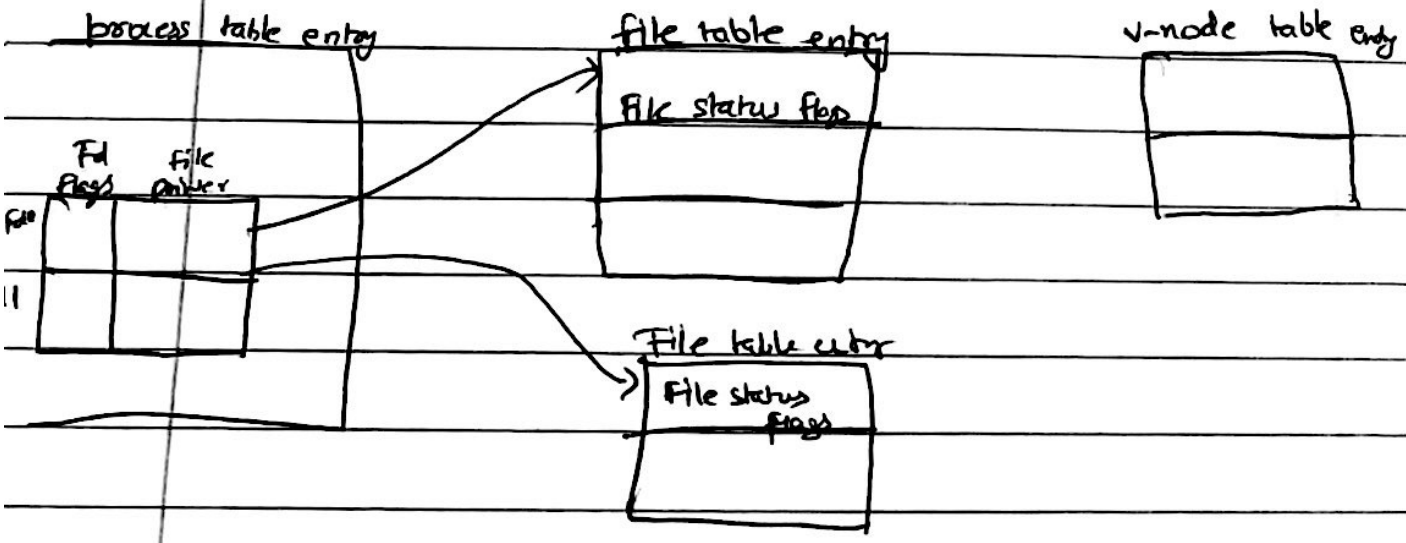
Advanced programming in Unix Environment

⇒ FILE SHARING

Data structure used by kernel for all I/O
Basically these are the three different tables
process table entry
file table entry
V-nodes table entry

it looks like:



In case of two independent processes have the same file open then we can have something like where two processes sharing the same V-node entry but having different file table entry with their own specification in file table entries like offset of the processes.

File descriptor flags & file status flags scopes.

- only to a single descriptor in a single process
- apply to all descriptors in any process that point to the given file table entry.

Everything works fine as long as we are reading the same file by multiple processes, but in case of writing we need to look out for strange errors.

In general, the term atomic operation refers to an operation that might be composed of multiple steps. If the operation is performed atomically, either all the steps are performed or none are performed.

FILES & DIRECTORIES

→ stat, fstat, fstatat, and lstat

stat → returns a structure of information about the named file.

fstat → obtains information about the file that is already open on the descriptor fd.

lstat → is similar to stat, but when the named file is a symbolic link.

fstatat function provides a way to return the file statistics for a pathname relative to an open directory represented by the fd argument.

File types → Regular file.
Directory file
block special file → disk drives devices
character special file → file providing unbuffered I/O access in variable-sized units to devices.
FIFO → type of file for communication between processes.
Socket → type of file used for network communications between processes
Symbolic link → file that points to another file.

file type is encoded in st_mode of stat structure.

S_ISREG()
S_ISDIR()
S_ISCHR()
S_ISBLK()
S_ISFIFO()
S_ISLNK()
S_ISSOCK()
→ file type macros in <sys/stat.h>

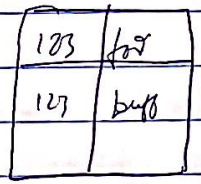
Set-User-Id & Set-Group-Id

→ Every process has six or more IDs associated with it.

real userID
real groupID
effective user ID
effective group ID
supplementary group IDs
saved set-user-ID
saved set-group-ID

→ means where we have a mapping of inode number to a file

Hardlink



123	foo
127	bar

Symbolic link

stat helps us to distinguish between symlinks

→ Why do we even need a symlink?
↳ in case of different file systems

homebrew on mac uses symlinks and internally switching file systems

link count → in ls -li filename
gives the number of files pointing at file.

ctime → write to struct stat
atime → read of data/file
mtime → write to file/data

"You cannot change the ctime, it will always read the current value"

* mkfifo /tmp/fifo
amazing things could be done through pipelines

→ read dir write dir

→ for hitting ls, we need the ls algorithm

open dir
read dir
print name

open dir
read dir
stat file (in case of symlink → use lstat)
print info

→ setid, then run → ls -l 'which ping'
setuid
setecuid → raises your priviliges

There's a huge difference between priviliges on directories v/s file

If effective uid → 0, you will get access regardless of permissions on the file

umask

→ drwxrwxrwt ⟶

→ chmod 777 v/s chmod 1777

→ In unix you can increase size but then cannot decrease the size (if talking in memory)

Directory sizes (Experiment with it)

* Shell-builtin (Know more about)

File system → inodes

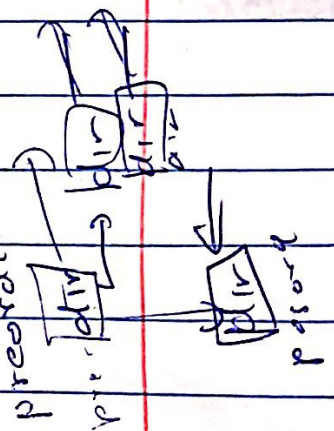
* Play with link counts
* for ls command
  → use opendir
  readdir
  rewinddir
  closedir

→ FTS. * a library

ls command

Dependent flags	Independent flags
a & A	d
b where a overwrites everything	F
h & K	-u
h overrides K	

-t -c



Compiler

stages

compiler chain

↳ pre processor

↳ Compiler

↳ assembler

↳ object code

↳ linker

→ gdb 10.exe → break main

→ run ~/test.s

fts

fts_ent

fts_read returns a pointer to an FTSENT structure describing a file in the hierarchy.

→ getting ftps

→ fts_open

hierarchy

A file can be visited in two ways. One is pre-order and another is post-order.

FTSENT & FTS are the two structures being defined in fts.h library.

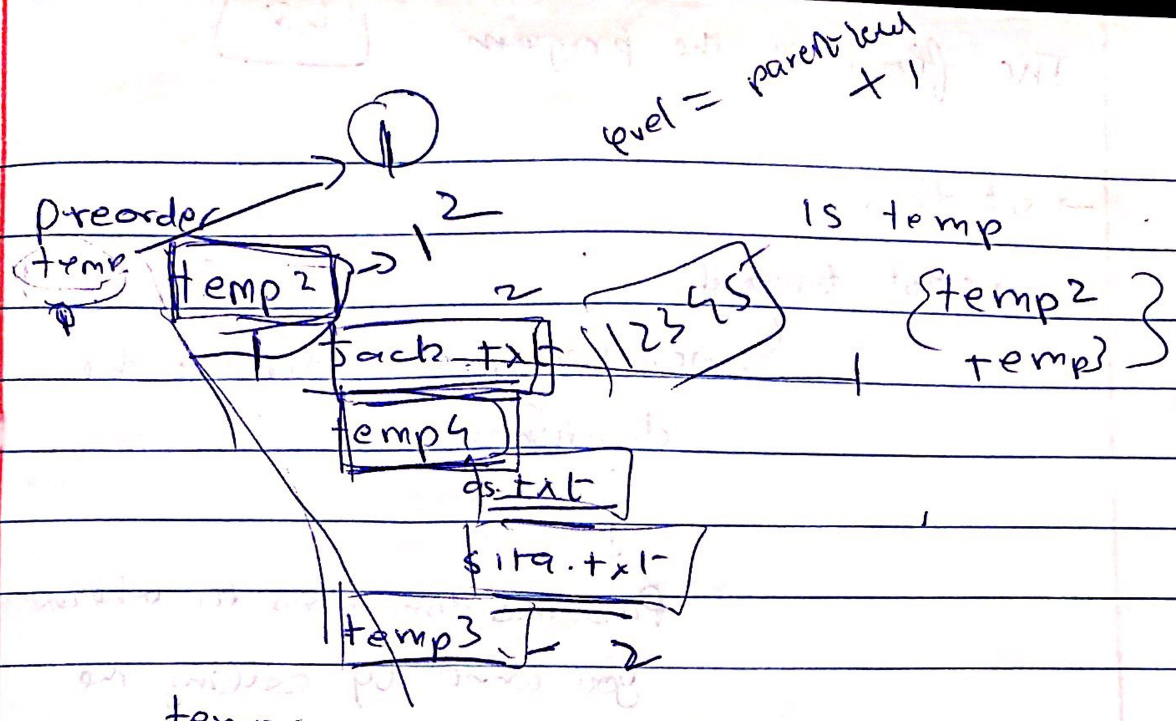
The useful stuff for us is FTS_D → for pre-order

FTS_DP → for post-order

FTS_F → a regular file

FTS_SL : a symbolic link

FTS_SLNONE : ... with non-existing target



temp:
↳ | temp2
temp3 |

ftr_read( )
fts_ch

tem/
/temp:
temp2 }
temp3 }

----1
----2

/temp/ temp2:
=
=
=

temp/temp3:
=
=
=

ne flow of the program (15)

set flags

→ start traversal

↳ now here first calculate the dimensions

↓

Pass this dimensions to wherever you want by calling the dimensions structure

call the actual printing traversal

in traversal call sorting functions to sort in FTS_open

for sorting functions

↳ we can use function poi to override the sorting o over each other