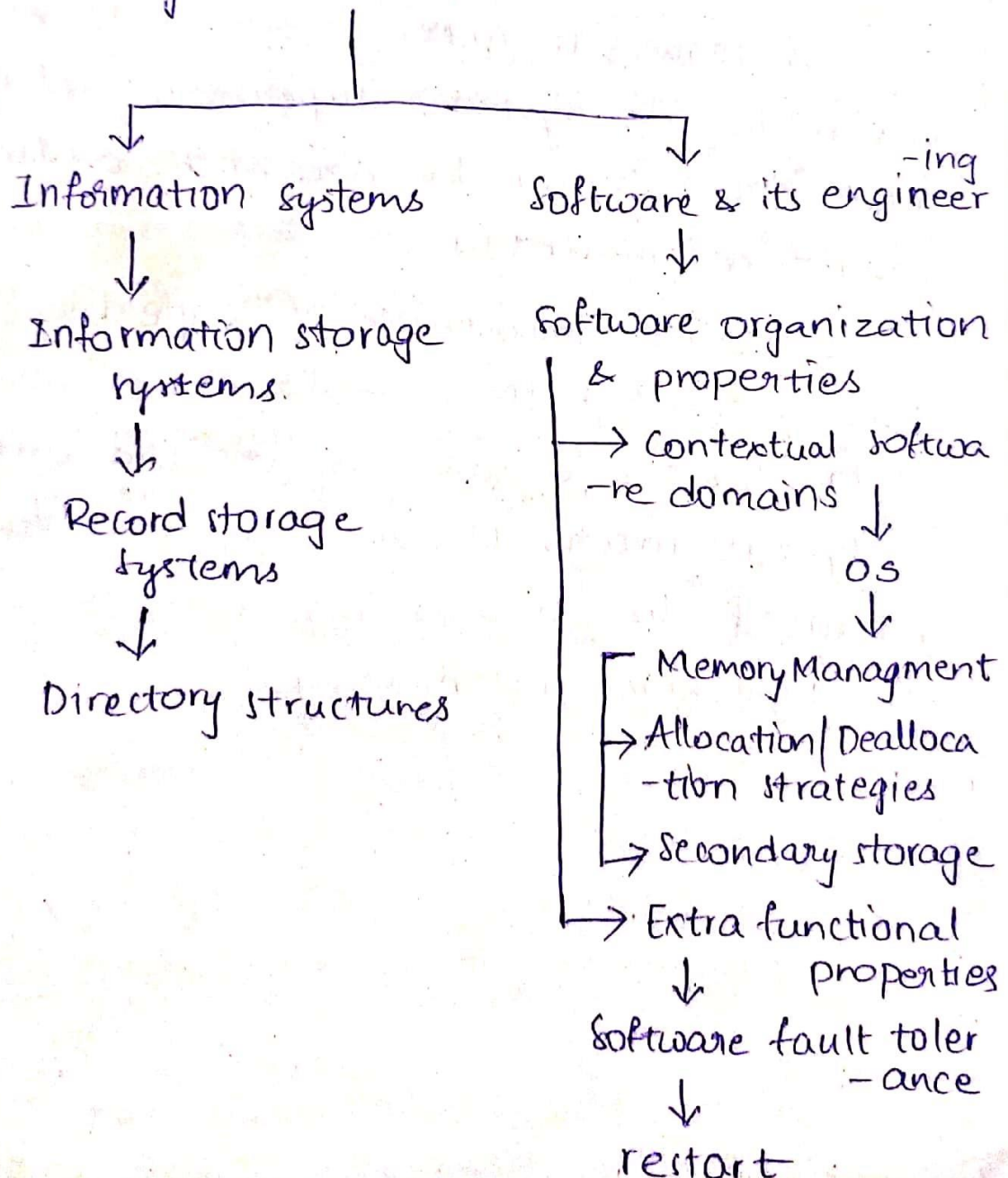


1A) Meta data updates, such as file creator, and block allocation, have consistently been identified as a source of performance integrity, security, & availability problems for file systems.

Log design:-

design & implementation of a log-structure
-d file system



→ In case of system crash, the system will be able to determine all metadata updates that not finish

A superblock is a record of a filesystem, including its size, the block size, the empty or filled blocks their respective counts, the size & location of inode tables, the disk block map & usage information, or size of block groups.

A request to access any file returns access to file systems superblock. If its superblock cannot be accessed, a file system cannot be mounted

Because importance of superblock & because damage to it, could erase crucial data, backup copies are created automatically at intervals on filesystems. For each mounted, linux also maintains a copy of its superblock in memory.

2A) you are not allowed to create a hard link to a directory. Each directory inode is allowed to appear once in exactly one parent directory & no more. The restriction means that every sub-directory only has one parent directory, that means special name ".." (dot dot) in a sub-directory always refers to its unique parent directory.

3A) (i) Directories map file systems names to inode numbers for you. Each inode is identified by a unique inode number that can be shown the `ls -li` command. Unix directories are what map file systems names to inode numbers that contains actual data.

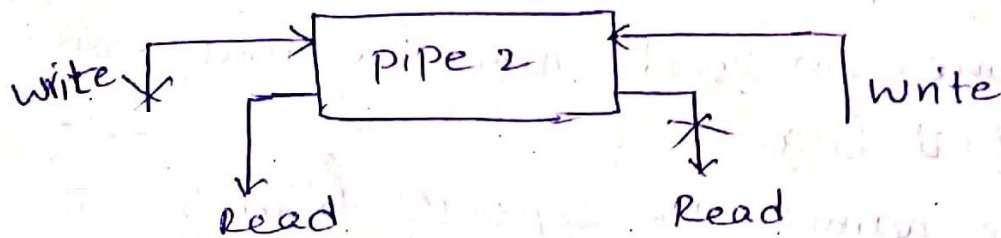
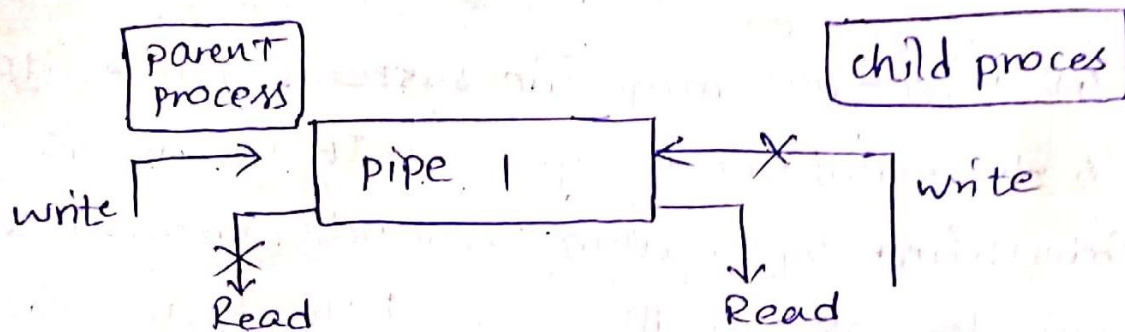
The names are separate from the things they name. When you access program by name, the system finds the programme in a directory, passed with inode num 266327 that holds actual data, & then the system has to go elsewhere on disk to inode numbers 266327 is actually access data for prog name.

File & directory data is actually stored under inode num, not under names.

3A)ii) Reading from small local file (a/b/c involves 4 separate disk operations

- (1) Reading disk containing root dir/
- 2 & 3 reading in disk block containing the directones b and c & reading in disk block containing the file c

4A) pictorial form of two way pipes



program for two way pipe.c

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

```
# include <unistd.h>
```

```
int main () {
```

```
    int pipefds1[2], pipefds2[2];
```

```
    int returnstatus1, returnstatus2;
```

```
    int pid;
```

```
    char pipe1writeMessage[20] = "Hi";
```

```
    char pipe2writeMessage[20] = "Hello";
```



```

char readmessage[20];

returnstatus1 = pipe ( pipefds1 );
if ( returnstatus1 == -1 ) {
    printf ( "unable to create pipe1 \n" );
    return 1;
}

returnstatus2 = pipe ( pipefds2 );
if ( returnstatus2 == -1 ) {
    printf ( "unable to create pipe2 \n" );
    return 1;
}

pid = fork ( ) ;
if ( pid != 0 ) // parent process
{
    close ( pipefds1 [0] );
    close ( pipefds2 [1] );
    printf ( "In parent : write to pipe 1-  
message is %s \n", pipe1writemessage );
    write ( pipefds1 [1], pipe1writemessage,
            sizeof ( pipe1writemessage ) );
    read ( pipefds2 [0], readmessage, sizeof
           ( readmessage ) );
    printf ( "In parent : Reading from pipe 2-  
message is %s \n", readmessage );
} else { // child process
    close ( pipefds1 [1] );
    close ( pipefds2 [0] );
    read ( pipefds1 [0], readmessage,
           sizeof ( readmessage ) );
}

```

```

printf ("In child : Reading from
        pipe message is '%s\n", readmessage);

printf ("In child : writing to pipe 2 - message
        is '%s\n", pipe2wntemessage);
write (pipefds2[1], pipe2wntemessage,
        sizeof (pipe2wntemessage));
}
return 0;
}

```

compilation: gcc two-way-pipe.c

execution/output:

In parent: writing to pipe 1 - message is Hi

In child: Reading from pipe 1 - message is Hi

In parent: writing to pipe 2 - message is Hello

In parent: Reading from pipe 2 - message is Hello

5A) ~~xv6~~ gives each process its own table of open files, as we saw, each open file is represented by struct file (3650), which is a wrapper around either an inode or a pipe, plus an i/o offset.

.\$ echo > a

log write 4 ialloc (44, from create 54)

log write 4 iupdate (44, from create 54)

log write 29 writei (47 from dirlink 48,

log write 2 iupdate from create 54)

6A) Algorithm for Allocating Disk block :-

Input :- No. of filesystem

output :- New block buffer

```
{
  while (superblock is locked)
  {
    sleep (event superblock not locked) ;
    remove block from superblock free list ;
    if (last block is removed from free list)
    {
      lock superblock ;
      read block which is just taken from freelist
      (algo bread)
      copy block number into superblock ;
      release block buffer (algo brelse) ;
      unlock the superblock ;
      wakeup processes (event superblock not locked) ;
    }
    get buffer from superblock list for removing
    zero buffer contents;      block (algo getblk);
    Decrement total count of free block;
    mention the superblock as modified;
    return buffer;
  } }
```


7A) (i) The process table entry contains a table, the file descriptor table that gives the mapping b/w the descriptor the process uses to refer to a file connection & and the data structure inside the kernel that represents the actual file connection.

In traditional implementation of UNIX, file descriptors index into a pre-process file descriptor table maintained by kernel, that in turn indexes into a system wide tables of files opened by all processes called the file-table. This table records mode with which file has been opened: for reading, writing, appending, & possibly other modes. File descriptor Table: There is a file descriptor table per process in U-Area. It contains A pointer to a file table entry.

several file descriptors are opened for every process: (i) stdin, (ii) stdout (iii) stderr
It contains OPEN-MAX entries, which must be atleast as big as POSIX_OPEN-MAX.

FileTable:- This is one FileTable in kernel.

These entries are pointed to by File descriptors & in turn point to file ID's :-

→ Count of no. of file descriptors pointing to this entry.

→ access mode : read or write

→ file ID :- (called inode in UNIX)

→ Current offset into the file

→ Access mode contains one of following:

O_RDONLY : open file with read only access.

O_WRONLY : open file with write only access

O_RDWR : open file with both read, write access

Generally unique to each process, but they can be shared by child processes created with fork subroutine or copied by the fcntl dup and dup2 subroutines.

File descriptors are indexes to file descriptors table in u-block area maintained by kernel for each process. The most common ways for process to obtain file descriptors through open or creat. When fork occurs, the description table is copied for child process, which allows child process equal to files used by parent process.

(ii) `open("/a/b/c", O_RDWR, O_CREATE)`

4851 : `sys-open`

4801 : `create`

4807 : `namei-parent [4396]` is slightly
diff : get inode for parent dir,
keep name to create

4811 : `dirlookup [4212]` checks if already
exists. `sys-open`, `sys-mknod`,
`sys-mkdir`.

4821 : get a fresh inode

4831 : add a name to directory,
`dirlink [4252]`

4832 : if `mkdir("/a/b/c")`, create
"." and ".." entries

4846 : return newly created inode

8A) Each direct pointer points to 4kB of
data, for a total of $10 \times 4 \text{ kB}$

The indirect pointer points to an
indirectly block, that contains pointer
to data. The indirect block is a 4kB
filled with 4-Byte pointers, for a total
of $1024 \times 4 \text{ kB}$ of data

Similarly, the double-indirect pointer points a double-indirect block containing 1024 pointers to indirect block, each of which points to 4KB * 1024 for a total of $1024 * 1024 * 4 \text{ KB}$ of data.

Finally, the triple-indirect pointer refers to $1024 * 1024 * 1024 * 4 \text{ KB}$

∴ The total data is thus, $10 * 4 \text{ KB} + 1024 * 4 \text{ KB} + 1024 * 1024 * 4 \text{ KB} + 1024 * 1024 * 1024 * 4 \text{ KB}$

9A) The read system call takes three arguments - (1) file descriptor of file.

(2) Buffer where the read data is to be stored

(3) the no. of bytes to be read from file.

A program needs to access the data from a file stored in a file system uses the read system call. The file is identified by a file descriptor that is normally obtained from a previous call to `open` or `read()`

`read()` is used to access data from a file that is stored in file system. The file to read can be identified by its file descriptor & it should be opened using `open()` before it can be read.

(ii) Yes, it is possible to read the desired block directly into user's buffer, & save overhead of copying.

10A) (i) Most of filesystems have method to assign permissions or access rights to specific users & groups of users. These permissions controlled the ability of users to view, change, Navigate and execute the contents of file systems. Menu options or functions may be made hidden depending on user's permission.

Each file is stored in a directory, & uses a directory entry that describes its characteristics such as its name, file extension & size. If at a dos prompt you type "Dir" to list files in directory by default you will not see any files that have "hidden" attribute set. While use of attributes is strictly "voluntary" they are important in certain circumstances. Deleting a Non-empty directory could work somehow as deleting an empty directory. by removing the pointer to directory's metadata there would be no pointers to item it contained, effectively

deleting all its children recursively.

(ii) dirent [3203]

- name : str
- inum : mode num for file
- empty directory entries : zeroed-out-inode

naming functions

- -namei [4354] does most name-to-inode translation
- skiplen [4314] parses path names
- dirlookup [4212] does name-to-inode lookup in single dir
- dirlink [4252] adds names to a directory

00044 // Directory is file containing a sequence of dirent structure

00045 # define DIRSIZE 14

00046

00047 struct dirent {

00048 ushort inum;

00049 charname [DIRSIZE]