

# **ADVANCED OPERATING SYSTEM**

Dr. Dipali Meher  
[mailtomeher@gmail.com](mailto:mailtomeher@gmail.com)

# AGENDA

Buffer Headers

Structure of Buffer Pool

Scenarios for retrieval of a buffer

Reading and writing a disk block

Inodes (accessing, releasing)

Structure of a regular file

Directories

Various system calls for file system

Pipes

Dup

Mounting and un-mounting file system

File sharing(atomic operations, stat,fstat and lstat functions)

File types

Maks and unmarks function

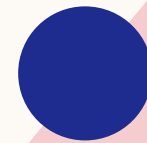
Chmod and fchmod functions

Sticky bit

Link, unlink, remove and rename functions

Mkdir and rmdir functions

Advanced file I/O





File is a logical storage unit which is collection of bytes, words or structures.

Each file has its own set of attributes(metadata) such as file name, size, location on disk, time of access, creation, modification and protection(owner of file)

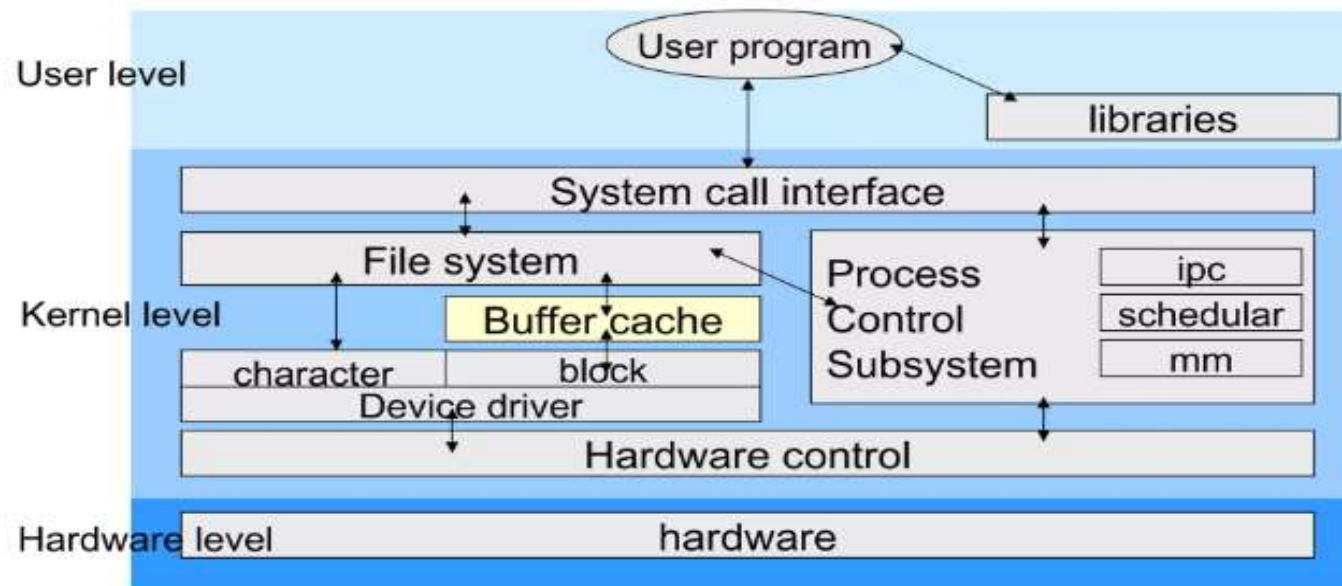
Buffer Headers:

The files are stored on hard drive and the processes can access these files and create new files on disk.

When a process request for a file the kernel brings the file into the main memory where user process can change read or access the file.

## Why Buffer cache

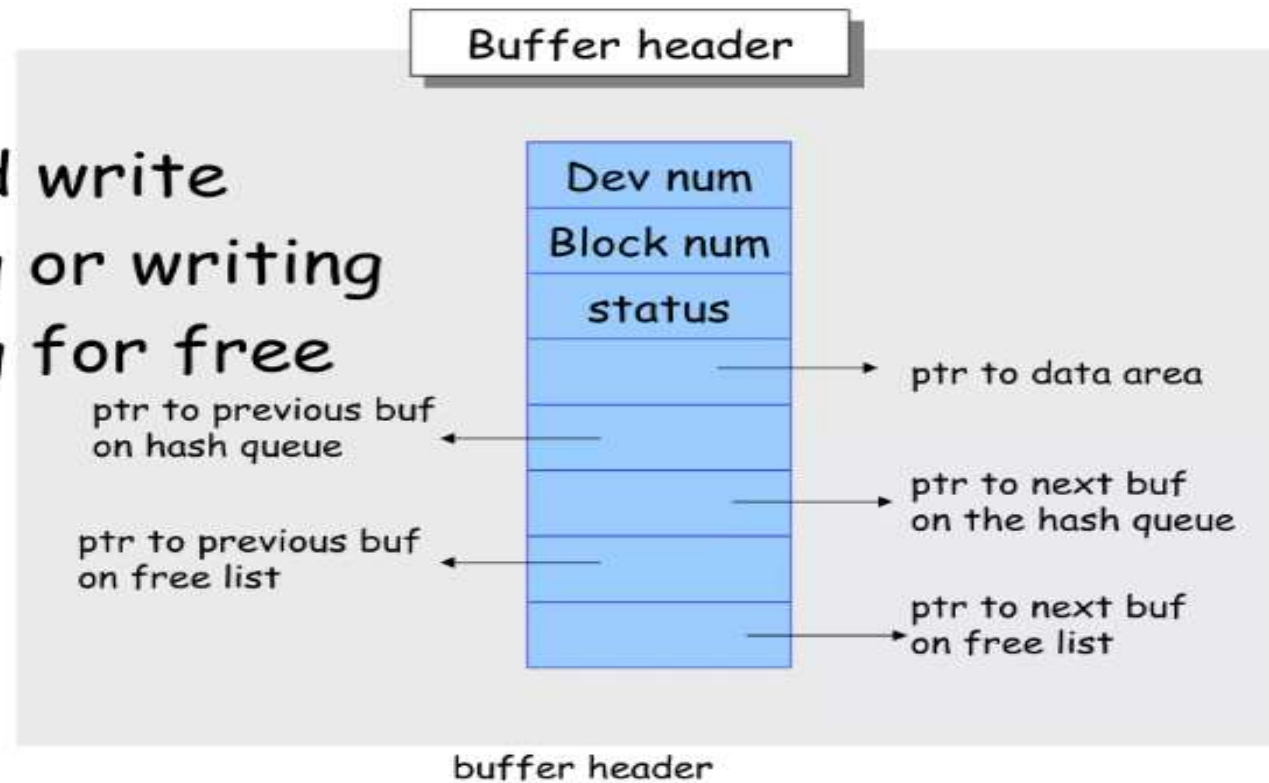
- ❑ To reduce the frequency of Disk Access
- ❑ Slow disk access time



block diagram of the System Kernel

# States of buffer

- ☐ Locked
- ☐ Valid
- ☐ Delayed write
- ☐ Reading or writing
- ☐ Waiting for free



# Structure of the Buffer Pool

---

## ❑ Free List

- Doubly linked circular list
- Every buffer is put on the free list when boot
- When insert
  - Push the buffer in the head of the list ( error case )
  - Push the buffer in the tail of the list ( usually )
- When take
  - Choose first element

## ❑ Hash Queue

- Separate queues : each Doubly linked list
- When insert
  - Hashed as a function device num, block num

# Scenarios for Retrieval of a Buffer

## ❑ To allocate a buffer for a disk block

- Use getblk()
  - Has Five scenarios

**algorithm getblk**

**Input:** file system number

**block number**

**Output:** locked buffer

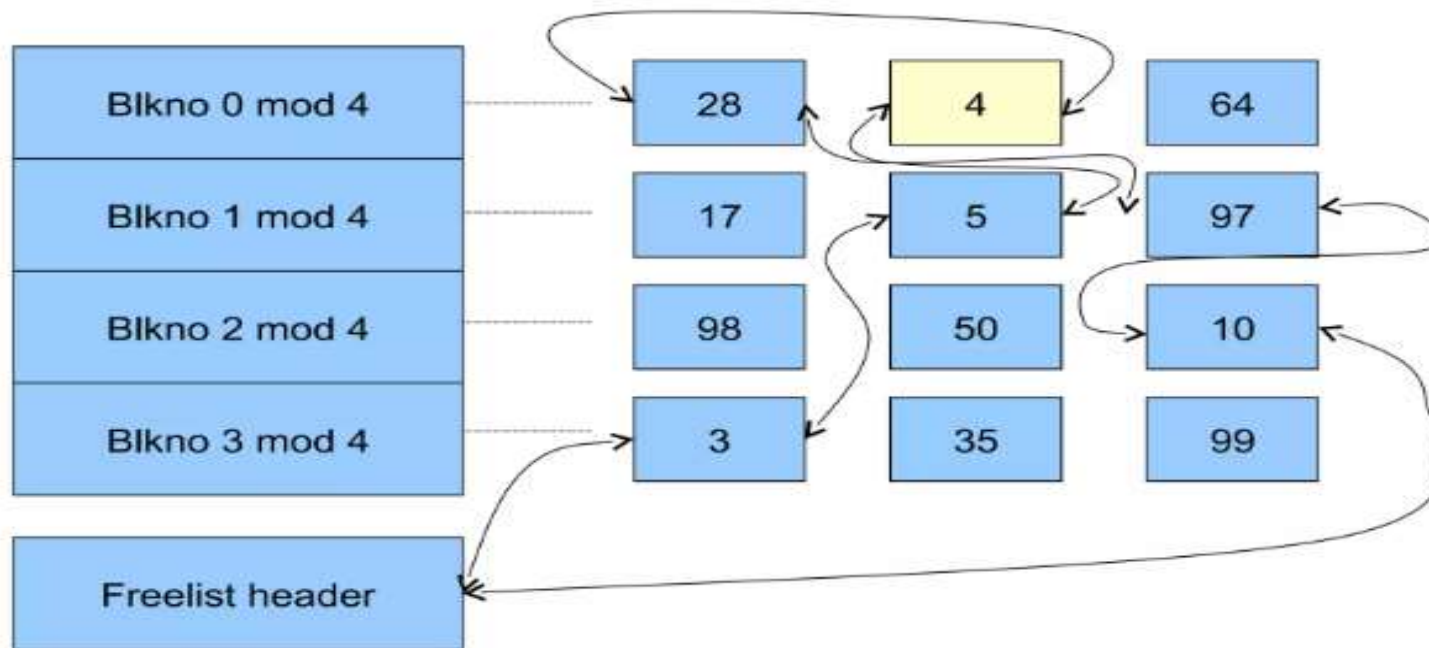
**that can now be used for block**

```
{
  while(buffer not found)
  {
    if(block in hash queue){ /*scenario 5*/
      if(buffer busy){
        sleep(event buffer becomes free)
        continue;
      }
      mark buffer busy; /*scenario 1*/
      remove buffer from free list;
      return buffer;
    }
  }
}
```

```
else{
  if ( there are no buffers on free list)
  {
    /*scenario 4*/
    sleep(event any buffer becomes free)
    continue;
  }
  remove buffer from free list;
  if(buffer marked for delayed write)
  {
    /*scenario 3*/
    asynchronous write buffer to disk;
    continue;
  }
  /*scenario 2*/
  remove buffer from old hash queue;
  put buffer onto new hash queue;
  return buffer;
}
}}
```

# 1<sup>st</sup> Scenario

- ❑ Block is in hash queue, not busy
- ❑ Choose that buffer

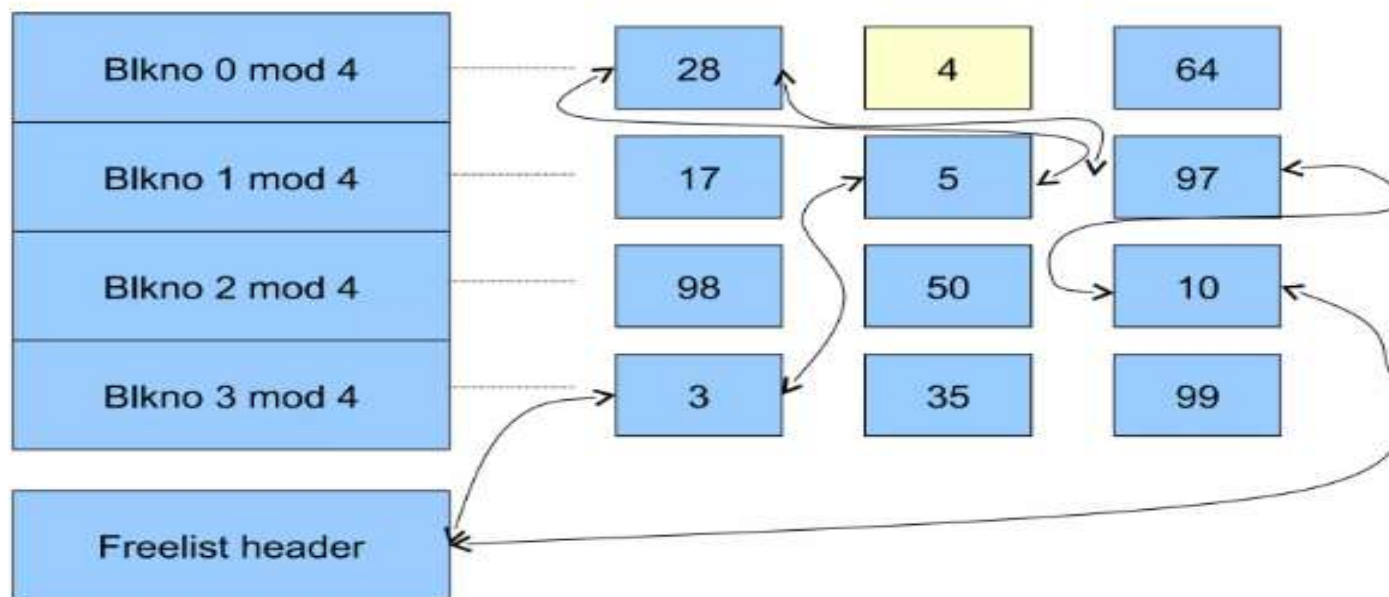


(a) Search for block 4 on first hash queue



# 1<sup>st</sup> Scenario

❑ After allocating

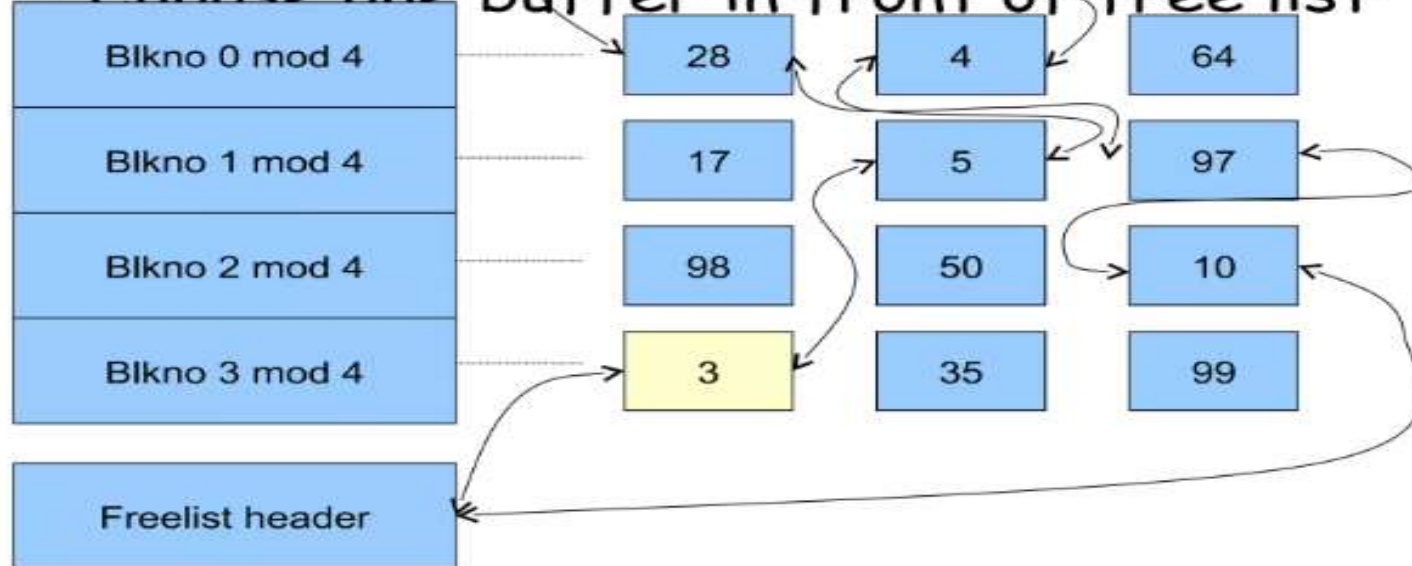


(b) Remove block 4 from free list

## 2<sup>nd</sup> Scenario

❑ Not in the hash queue and exist free buff.

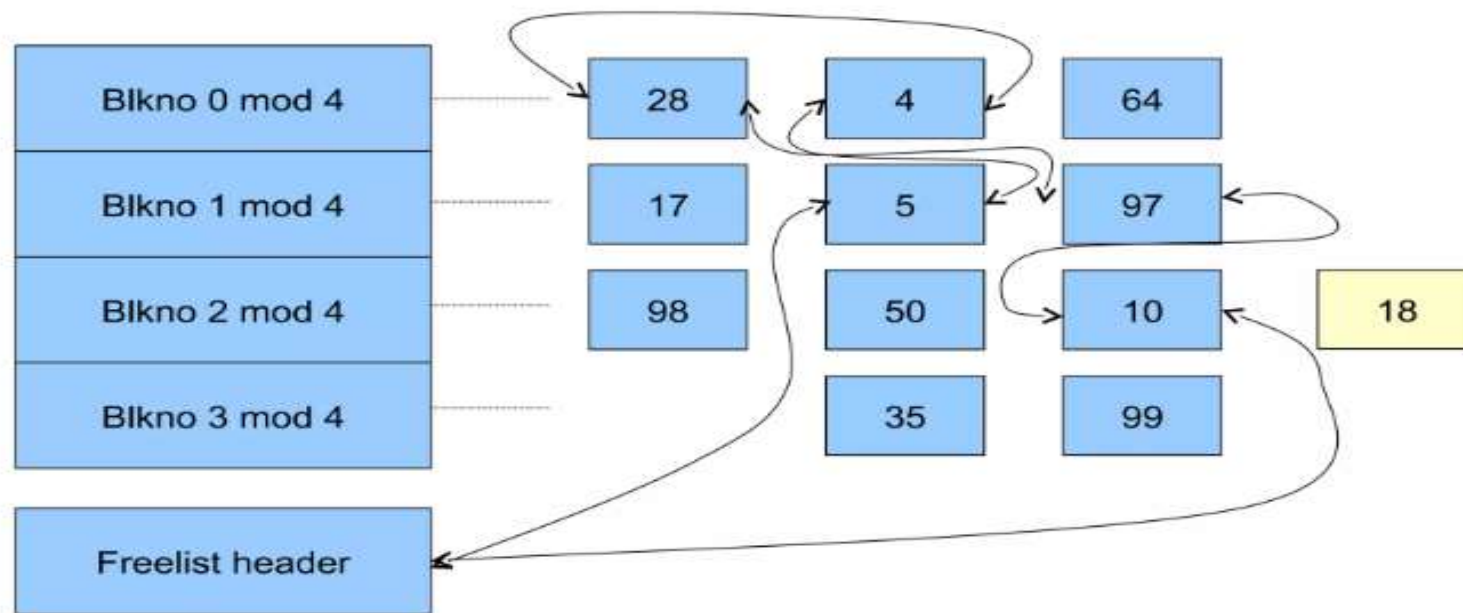
▪ Choose one buffer in front of free list



(a) Search for block 18 - Not in the cache

## 2<sup>nd</sup> Scenario

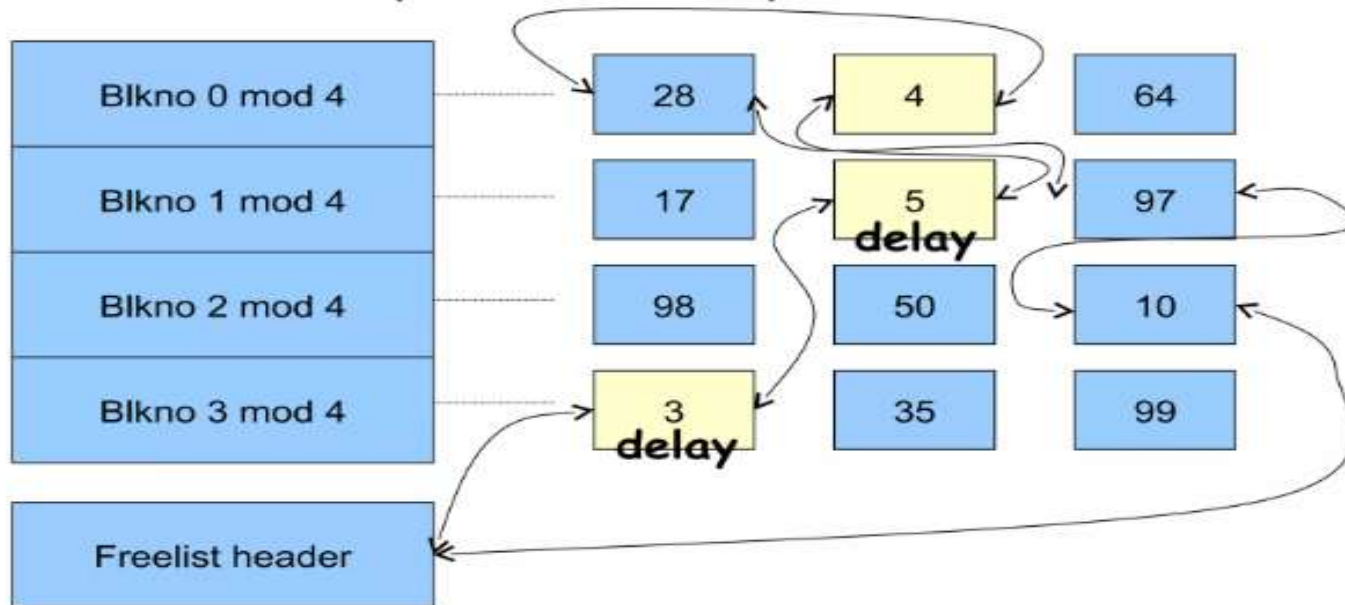
❑ After allocating



(b) Remove first block from free list, assign to 18

### 3<sup>rd</sup> Scenario

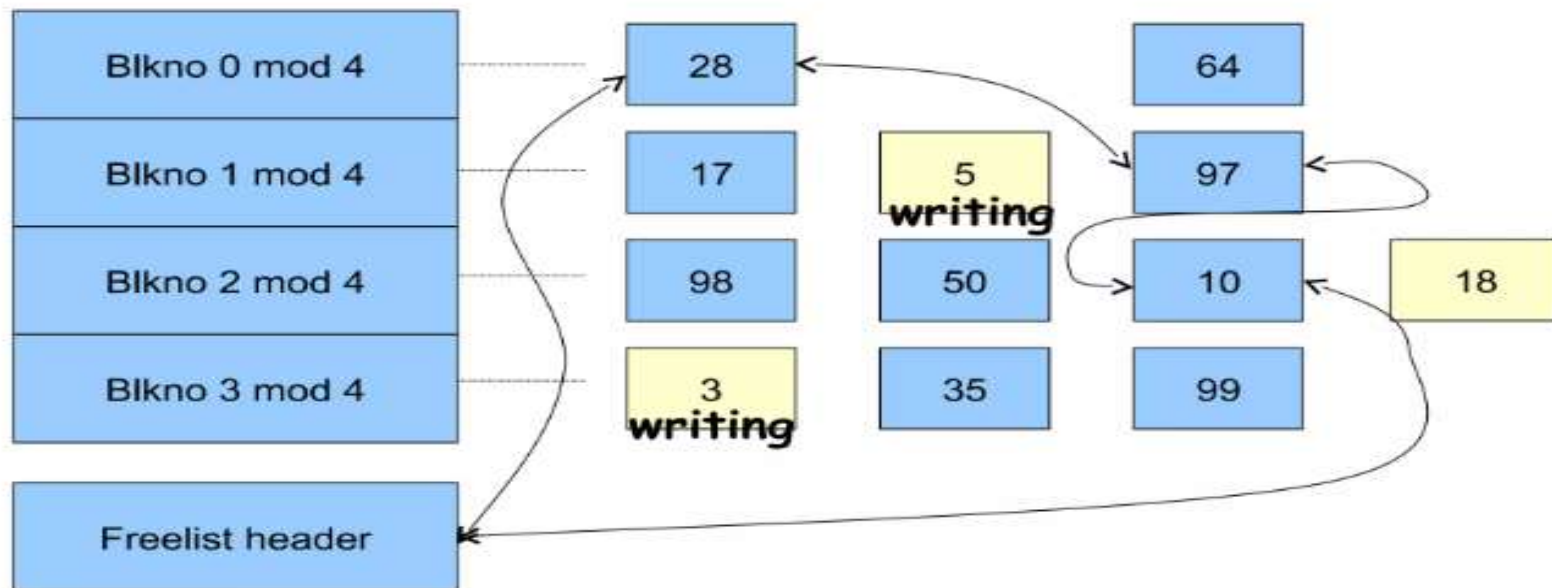
- ❑ Not in the hash queue and there exists delayed write buffer in the front of free list
  - Write delayed buffer async. and choose next



(a) Search for block 18, delayed write blocks on free list

## 3<sup>rd</sup> Scenario

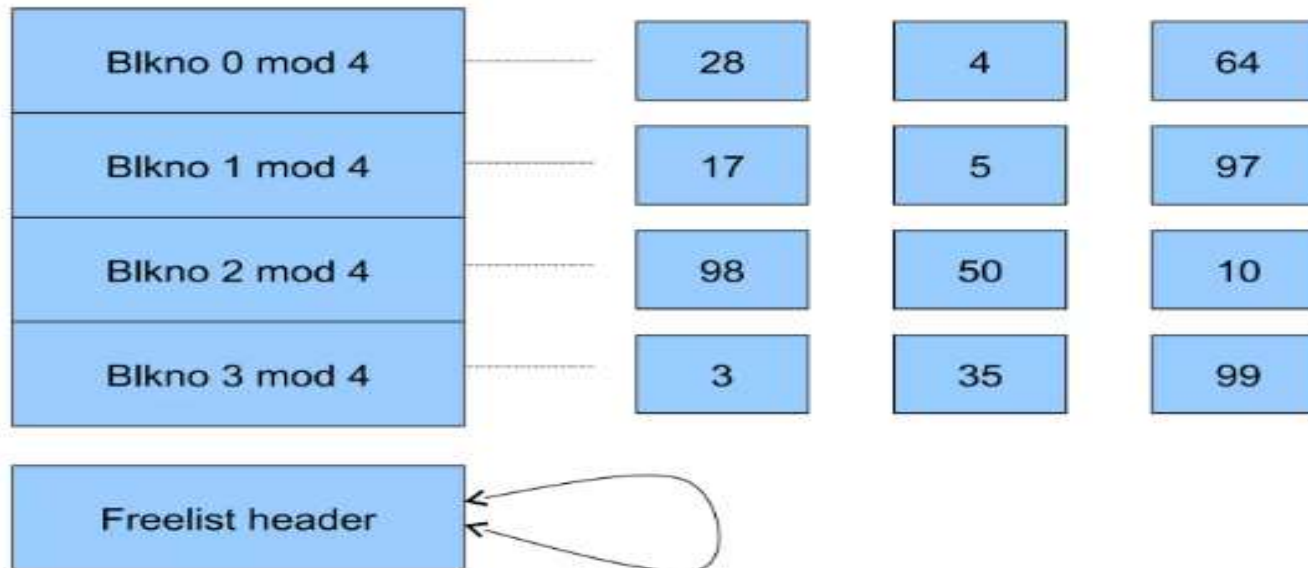
### ❑ After allocating



(b) Writing block 3, 5, reassign 4 to 18

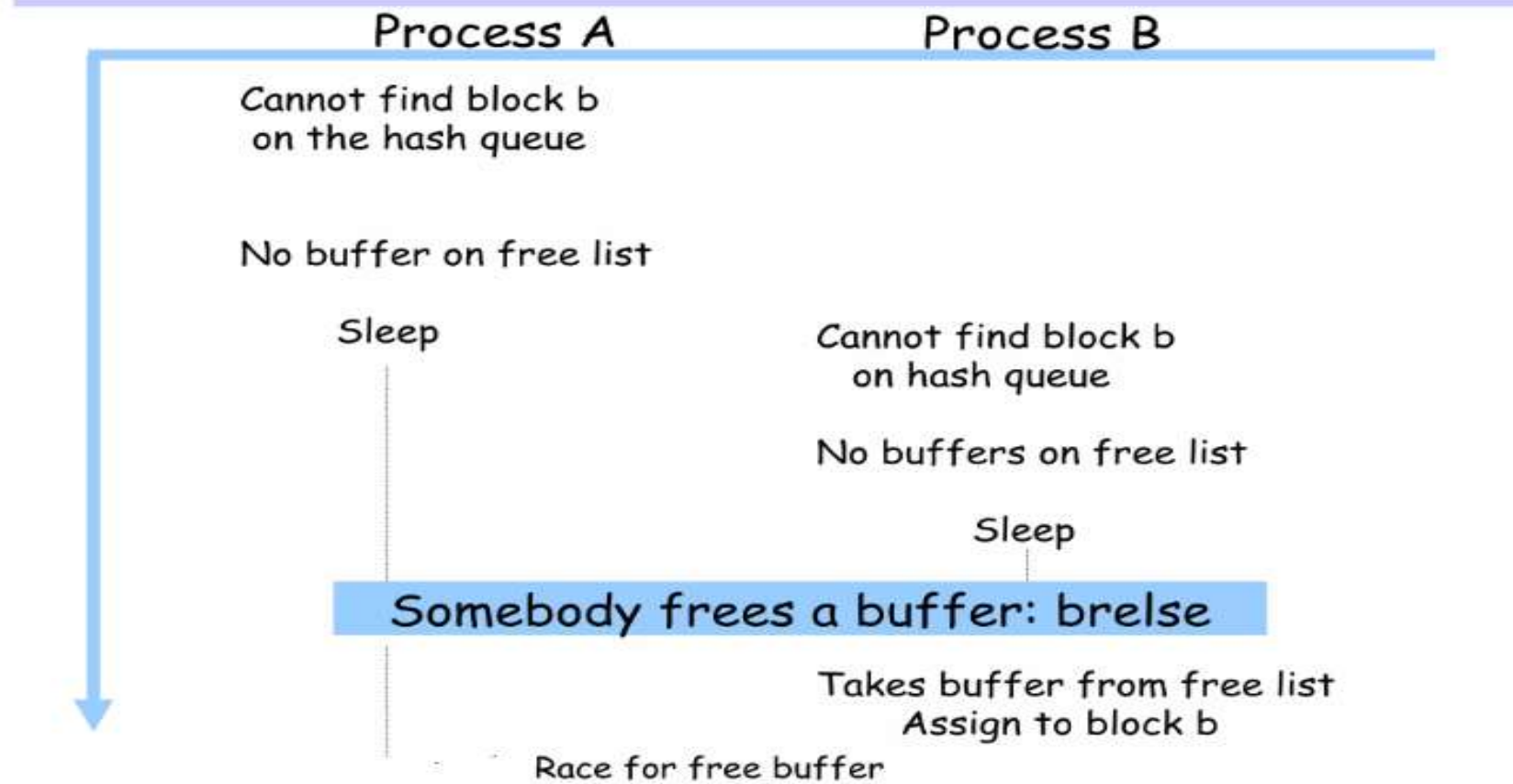
## 4<sup>th</sup> Scenario

- ❑ Not in the hash queue and no free buffer
  - Wait until any buffer free and re-do



(a) Search for block 18, empty free list

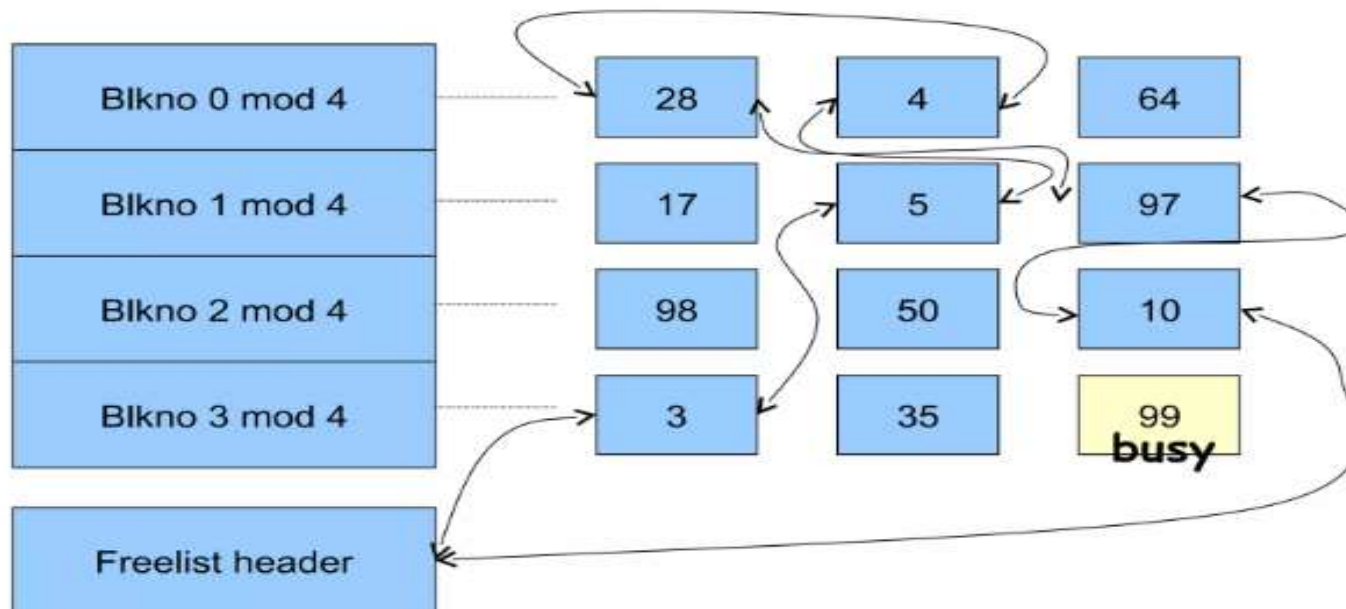
## 4<sup>th</sup> Scenario





## 5<sup>th</sup> Scenario

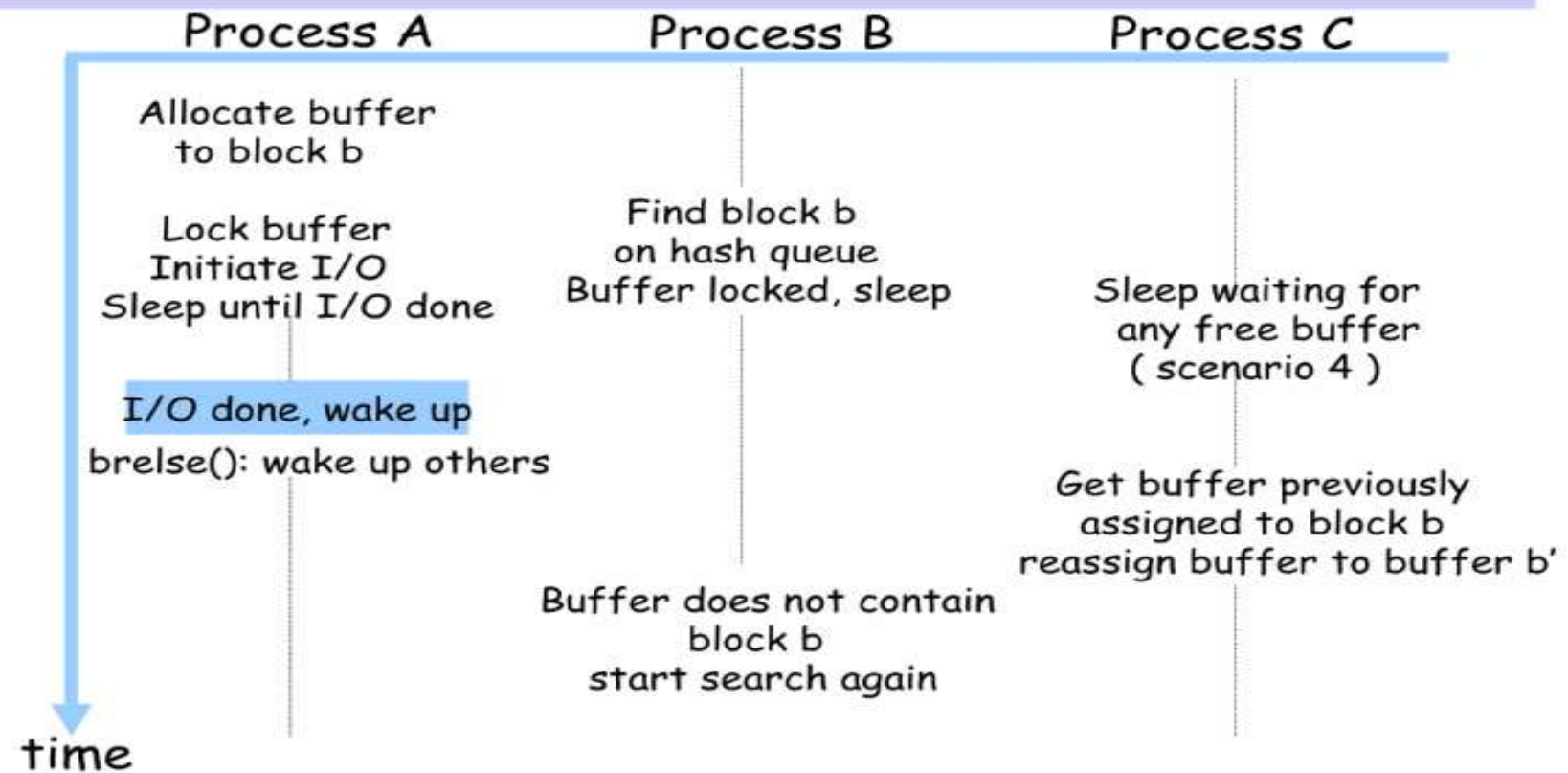
- ❑ Block is in hash queue, but busy
  - Wait until usable and re-do



(a) Search for block 99, block busy



## 5<sup>th</sup> Scenario



## Reading and Writing Disk Blocks

Algorithm: bread

Input: file system number block number

Output: buffer containing data

```
{  
  get buffer for block (algorithm: getblk);  
  if (buffer data valid) return buffer;  
  initiate disk read;  
  sleep (event: disk read complete);  
  return buffer;  
}
```

## Reading and Writing Disk Blocks

19

Algorithm: bwrite

Input: buffer

Output: none

```
{ initiate disk write;
if (I/O synchronous)
{
sleep (event: I/O complete);
release buffer (algorithm: brelse);
}
else if (buffer marked for delayed write)
mark buffer to put at head of free list;
}
```

# Inodes

20

Inode stands for Index Node.

*An inode is a data structure ... .. that stores all the information about a file except its name and its actual data.*

Inodes stores metadata about the file it refers to.

This metadata contains all the information about the said file.

- Size
- Permission
- Owner/Group
- Location of the hard drive
- Date/time
- Other information

1.	iget	→	Returns previously identified I node
2.	input	→	Releases an inode
3.	bmap	→	Sets kernel parameters for accessing a file
4.	namei	→	Converts a user level path name to an inode
5.	alloc	→	Allocates disk blocks to a file
6.	free	→	Free the disk blocks allocated to a file
7.	Ialloc and ifree	→	Assigns and free the inodes for files

## Accessing inodes

block number =  $\left( \frac{\text{inode number} - 1}{\text{number of inodes per block}} \right) + \text{start block of inode list}$  <sup>21</sup>

When higher level algorithms requests for inode, kernel identifies particular inode by their file system and inode number and allocates it in-core inodes. Kernel can identify inode using logical disk block which contains the inode depending upon how many disk inodes fit into a disk block which contains the inode using above formula.

For example

- i) Beginning of the inode list is block 2
- ii) There are 8 inodes per block

Then

$$\begin{aligned} \text{Block num} &= ((8-1)/(8)) + (2) \\ &= 7/8 + 2 = 0 + 2 \quad (\text{The inode number 8 is in disk block}=2) \end{aligned}$$

Another example

$$\begin{aligned} \text{Block num} &= ((9-1)/8) + 2 \\ &= ((8/8) + 2 = 1 + 2 = 3 \quad (\text{The inode number 9 is in disk blocks 3}) \end{aligned}$$

where the division operation returns the integer part of the quotient

# Releasing inodes

- ❑ Algorithm **iput** is used by kernel to release an inode
- ❑ The kernel decrements its inode reference count.
- ❑ If the count drops to 0 then kernel writes the inode to the disk
- ❑ The kernel places the inodes on the free list of inodes
- ❑ Kernel releases all data blocks associated with file and free inode number

# Structure of a Regular File

23

Inode: It is table of contents to locates a files data on disk .

TOC consists of set of disk block numbers as each block on a disk is addressable by number.

There are two versions of inode.

- 1) Disk copy: inode information when file is not in use
- 2) In-core-records information about active files.

Kernel will allocate contiguous space for file system(before allowing any operation that will increate file size)

	File A	File B	File C	
	40	50	60	70

Block Address

File A	Free	Free	File C	File B	
40	50	60	70	80	91

Kernel periodically run garbage collector to collect available free space

## Structure of a Regular File

24

Kernel allocates one block to a file at a time.

Processes access data in a file by byte offset.

(.i.e how many number of bytes required or file is stored in how many number of bytes). Starting byte address is 0.

Kernel converts the user view of bytes into view of blocks.

File starts at logical block 0 then contiguous block numbers are assigned to file till its size.

Kernel accesses the inode and converts the logical file block into the appropriate disk block



# Directories

25

A directory is a file whose data is sequence of entries, each consisting of an inode number and the name of a file contained in the directory. A path name is a null terminated character string is divided into separate components by the / character.

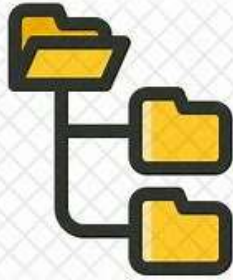
Last component is non directory file

Component name is 14 character ( 2 bytes inode number size of dir entry 16 bytes)

Processes read the directories as the same way they read regular files. Kernel reserves some rights to write a directory

Following are the rights/permission given by kernel to a directory

- 1) Read permission- allow process to read a directory
- 2) Write permission- allow process to create new directory, remove old( sys calls used are – create,mknod,link,unlink)
- 3) Execute permission- allow process to search directory for a filename



System calls is the way that program talk to the operating system.

File System Calls							
Return File Desc	Use of namei		Assign inodes	File Attributes	File I/O	File Sys Structure	Tree Manipulation
open creat dup pipe close	open creat chdir chroot chown chmod	stat link unlink mknod link mount umount	creat mknod link unlink	chown chmod stat	read write lseek	mount umount	chdir chown
Lower Level File System Algorithms							
namei							
iget	iput	ialloc	ifree	alloc	free	bmap	
buffer allocation algorithms							
getblk		brelease	bread	breada	bwrite		

## Open System call:

Syntax:

`fd=open(pathname, flags, modes)`

Pathname → filename

Flags →

type(read/write)

Modes → file permission if the file being created

Fd → file descriptor

```
/* Algorithm: open
 * Input: pathname
 *         flags
 *         mode (for creation type of open)
 * Output: file descriptor
 */

{
    convert file name to inode (Algorithm: namei);
    if (file does not exist or access is not permitted)
        return (error);
    allocate file table entry for inode, initialize count, offset;
    allocate user file descriptor entry, set pointer to file table entry;
    if (type of open specifies truncate file)
        free all blocks (algorithm: free);
    unlock (inode);
    return (user file descriptor);
}
```

## Read System call:

### Syntax:

number=read(fd,  
buffer, count)

fd → file  
descriptor

buffer → address  
of data structure  
in user process

for data reading

Count → no of  
bytes the user  
want to read

```
/* Algorithm: read
 * Input: user file descriptor
 *        address of buffer in user process
 *        number of bytes to be read
 * Output: count of bytes copied into user space
 */
{
    get file table entry from user file descriptor table;
    check file accessibility;
    set parameters in u-area for user address, byte count, I/O to user;
    get inode from file table;
    lock inode;
    set byte offset in u-area from file table offset;
    while (count not satisfied)
    {
        convert file offset to disk block (algorithm: bmap);
        calculate offset into block, number of bytes to read;
        if (number of bytes to read is 0)
            break;                // trying to read End Of File (EOF)
        read block (algorithm: bread or breada whichever applicable);
        copy data from system buffer to user address;
        update u-area fields for file byte offset, read count, address to write into user space;
        release buffer;           // locked in bread
    }
    unlock inode;
    update file table offset for next read;
    return (total number of bytes read);
}
```

## Write System call:

Syntax: `number=write(fd, buffer, count)`

`fd` → file descriptor

`Buffer` → address of data structure in user process that will contain the read data

`Count` → number of bytes user want to write  
inode is locked for write duration( as kernel may change inode number while allocating new block to file  
When write operation is complete kernel updates the file size entry in the inode of file grows larger)

## Close System call:

Syntax: `close(fd)`

`fd` → file descriptor

kernel closes file with inodes and file descriptors.

Create  
System call:  
Syntax:  
fd=create(path  
name,  
modes)  
path name→  
fd→ file  
descriptor  
modes→  
read/write

```
/* Algorithm: creat
 * Input: file name
 *      permissions
 * Output: file descriptor
 */

{
    get inode for file name (algorithm: namei);
    if (file already exists)
    {
        if (not permitted access)
        {
            release inode (algorithm: iput);
            return (error);
        }
    }
    else
    {
        assign free inode from file system (algorithm: ialloc);
        create new directory entry in the parent directory: include new file name and newly assigned inode number;
    }
    allocate file table entry for inode, initialize count;
    if (file existed at time of creat)
        free all file blocks (algorithm: free);
    unlock (inode);
    return (user file descriptor);
}
```

Collected By: Dr. Dipali Meher

Source: The Design of Univ Operating System- Maurice Bach

## Fsync, fdatasync function

**int fsync(int fildes)**

This function refers only a single file, specified by the file descriptor

fildes and waiting for the disk

writes of delayed write blocks to complete before returning

**int fdatasync(int fildes)**

Same as fsync. It affects only data portions of a file with fsync, the files

attributes are also updated synchronously



Adjusting the position of file I/O using 'lseek'

position = lseek(fd, offset reference)

fd: file descriptor

Offset is a byte offset.

Reference indicate whether offset should be considered from the beginning of the file from the current position of the read/ write offset or from end of file.

To implement this function kernel adjusts offset value in the file table;

subsequent read or write system calls use the file table offset as their starting byte offset.

# Pipes

34

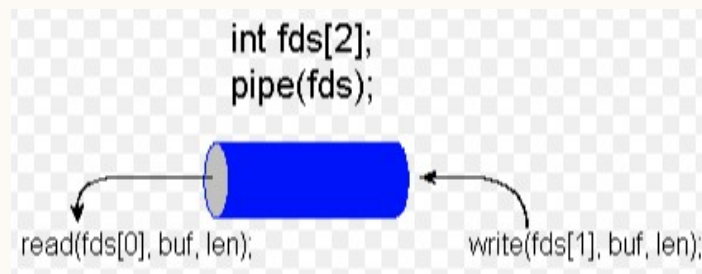
Pipes allow transfer of data between processes in FIFO manner and also allow synchronization of process execution.

Two types 1) Named pipes 2) Un-named pipes

Syntax for creating pipe is

`pipe(fdptr);`

`fdptr`: pointer to an integer array that will contain two file descriptors for reading and writing pipes.



<b>Named Pipes</b>	<b>Un-named Pipes</b>
Names pipes have a directory entry and path name	It lacks a directory entry and path name
It is initialized using open() system call	It is accessed using pipe() system call
Names pipes permanently exist in the file system hierarchy	The kernel retakes the inode assigned to unnamed pipes, they are transient

## Read Write operations of pipes

Read Write operations of pipes done in FIFO manner. Kernel access the data for a pipe as same as regular file. Kernel stores data on the pipe device and assign blocks to the pipe as needed during write calls

Pipe uses only the direct blocks of the inode for greater efficiency.

## Closing pipes

37

Kernel decrements the number of pipe readers or writers according to the type of file descriptor.

The kernel frees all data blocks of pipe and adjusts inode to indicate that pipe is empty.

# dup

This system call is used to make duplicate file descriptor<sup>38</sup> into the first free slot of the user file descriptor table returning the new file descriptor to the user.

Syntax

```
newfd=dup(fd)
```

Dup system call makes copy of file descriptor, it increments the count of the corresponding file table entry as it has one more file descriptor entry.

## Mounting and un-mounting file systems

39

These system calls connect and disconnect file system from the hierarchy.

The mount system call allows users to access data in a disk section instead of sequence of disk blocks.

Syntax:

mount(special pathname, directory pathname, options)

**Special pathname**: name of the special device file of the disk section containing the file system to be mounted.

**Directory pathname**: directory in the existing hierarchy where the file system will be mounted.

Option: indicate whether the file system should be mounted “read-only”

Example: mount(“dev/disk1”, “/user”, 0)

# Mounting and un-mounting file systems

40

Kernel has mount table with entries for every mounted file system where each mount table entry consists of the following items:

- i) A device number**
- ii) A pointer to buffer containing the file system super block**
- iii) A pointer to the root inode of mounted file system**
- iv) A pointer to the mount point**

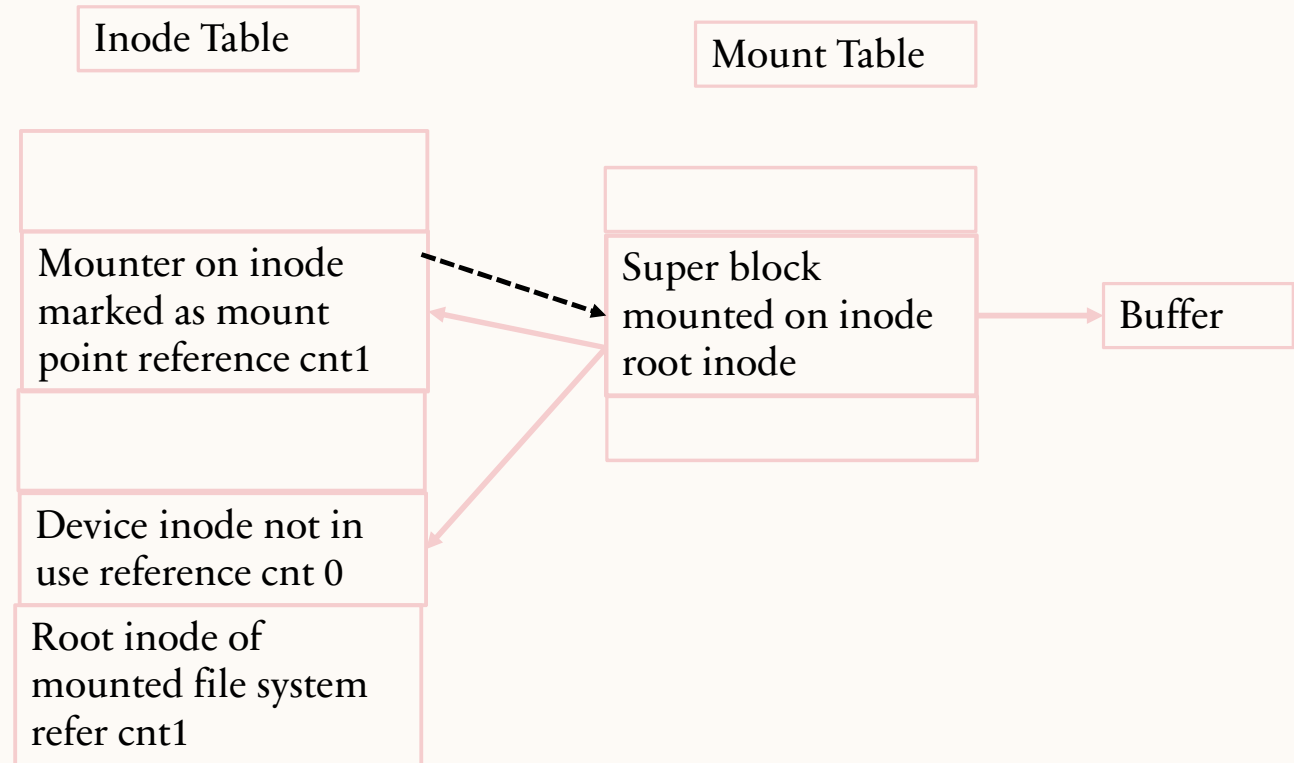


# Un- mounting file systems

The system call unmount is used to unmount file.<sup>41</sup>  
**The syntax is**  
**Unmount(special filename);**  
Special filename indicates the file system to be unmounted.

Kernel releases inode and finds the mount table entry having device number equal to that of special file.

# Data structures after mount



# File sharing

43

Unix system supports sharing of open files among different processes. The kernel used three data structures for file sharing.

- i) Every process has an entry in the process table.
- ii) The kernel maintains the file table for all open files
- iii) Each open file has v-node structure that contains information about the type of file and pointers to functions that operate on the file.

V-node : virtual file system by Sun microsystems.

## Atomic Operations

44

Atomic operation: refers to an operation that might be composed of multiple steps.

If the operation is performed atomically, either all steps are performed or non are performed.

They are

- 1)Appending to a file
- 2)Pread and pwrite system call
- 3)Creating a file

## Atomic Operations

45

Appending to a file: this operation is performed at a position to the end of file without O-APPEND option to open.

Example

```
if(lseek(fd,OL,2)<0)
Err_sys("lseek error");
If(write(fd,buf,100)!=100)
err_sys("Write error")
```

This code works for single file not multiple files. When an operation requires more than one function call then it is not atomic. In this situation kernel can suspend the process between two function calls

O-APPEND flag is set at time of file creation to perform atomic operations on files

Collected By: Dr. Dipali Meher

Source: The Design of Univ Operating System- Maurice Bach

## **pread() & pwrite() system call:**

46

These system calls allow seek I/O operations to be performed atomically.

pread and pwrite are same as lseek with following exceptions:

- a. The file pointer is not updated
- b. There is no way to interrupt the two operations using pread and pwrite

```
ssize_t pread(int fildes, void * buf, size_t n_bytes, off_t offset)
```

```
Ssize_t pwrite(int fildes, const void *buf, size_t nbytes, off_t offset);
```

## **pread() & pwrite() system call:**

47

These system calls allow seek I/O operations to be performed atomically.

pread and pwrite are same as lseek with following exceptions:

- a. The file pointer is not updated
- b. There is no way to interrupt the two operations using pread and pwrite

```
ssize_t pread(int fildes, void * buf, size_t n_bytes, off_t offset)
```

```
Ssize_t pwrite(int fildes, const void *buf, size_t nbytes, off_t offset);
```



## Creating a file:

open operation will fail if a attempt is made for creating a file which already exists.

This check for existence of the file and the Creation of the file are treated as atomic operations

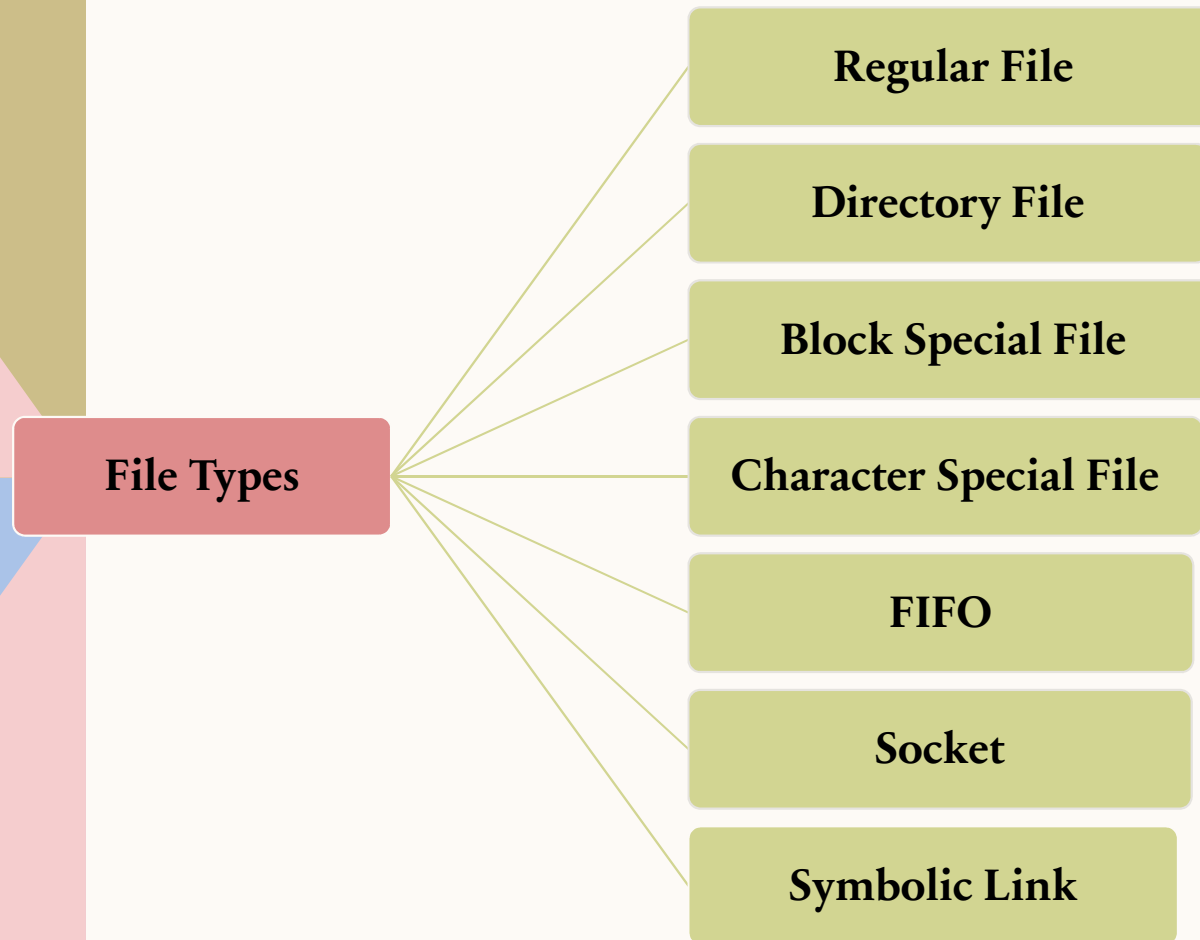


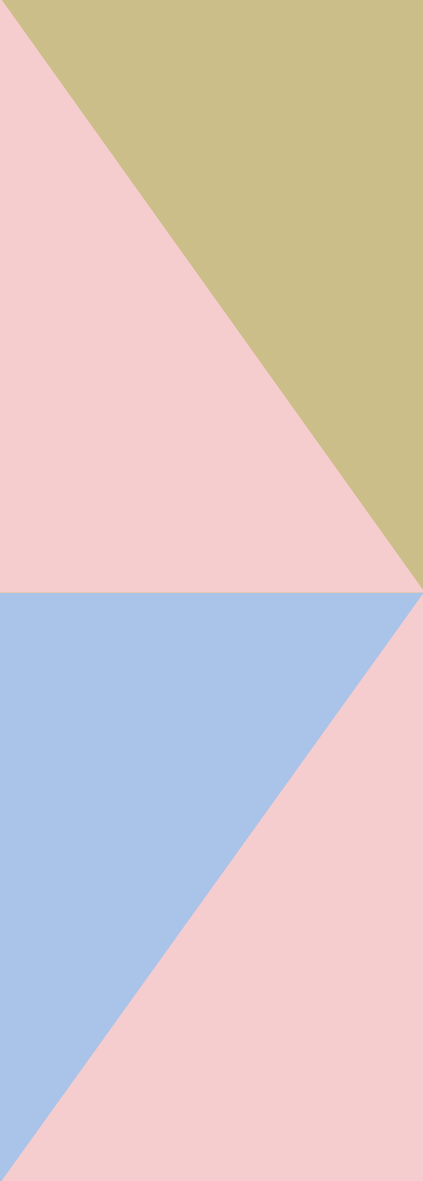
# Stat function

Stat function returns a structure of information about the named file, fstat function obtains information about the file that is already open on the descriptor.

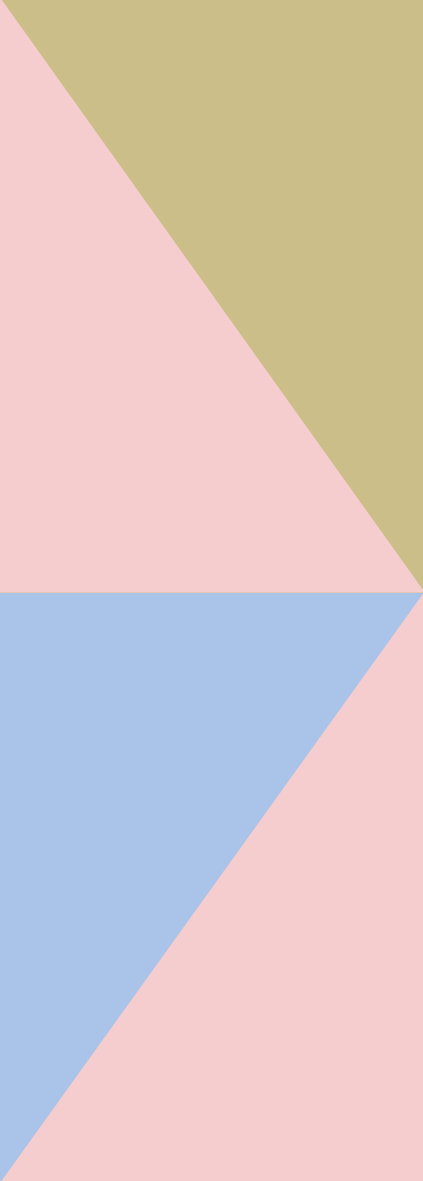
The lstat function is similar to stat but when the named file is symbolic link lstat function returns information about the symbolic link, not the file referenced by the symbolic link. The structure stat will be as follows:

```
struct stat {  
    dev_t st_dev; /* ID of device containing file */  
    ino_t st_ino; /* inode number */  
    mode_t st_mode; /* protection */  
    nlink_t st_nlink; /* number of hard links */  
    uid_t st_uid; /* user ID of owner */  
    gid_t st_gid; /* group ID of owner */  
    dev_t st_rdev; /* device ID (if special file) */  
    off_t st_size; /* total size, in bytes */  
    blksize_t st_blksize; /* blocksize for file system I/O */  
    blkcnt_t st_blocks; /* number of 512B blocks allocated */  
    time_t st_atime; /* time of last access */  
    time_t st_mtime; /* time of last modification */  
    time_t st_ctime; /* time of last status change */  
};
```





**Regular File:** Most common type of file,  
This file contains data ( UNIX kernel does not  
identify data(text/binary).  
Content interpretation has to be done by  
application itself.  
All binary files has to confirm with kernel to  
load whether file data  
or files text.



**Directory File:** A file that contains names of other files and pointers to information these files.

Process with read permission to directory file- read the contents of the directory.

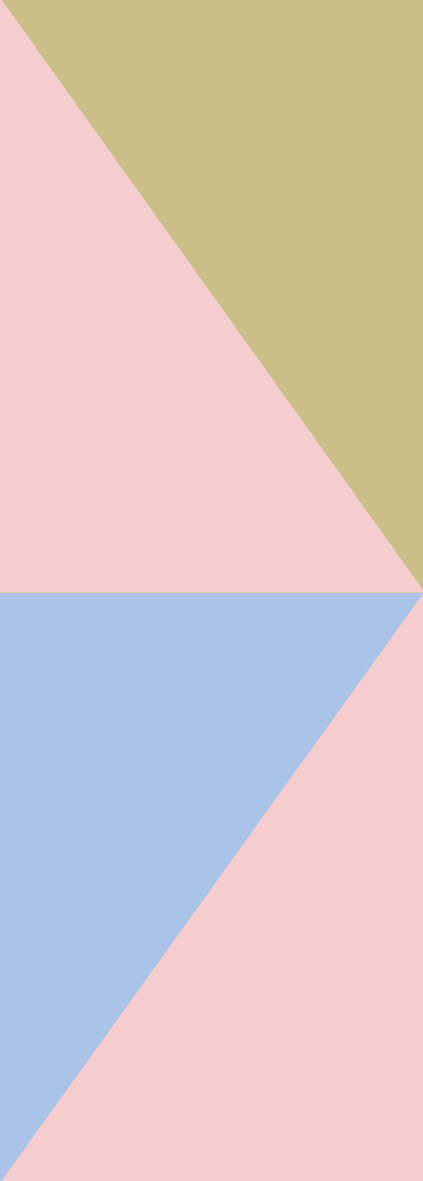
Kernel has permission to write to directory file,

Process use the system calls and functions to make change sin the directory.



**Block Special File:** This file type providing buffered I/O access in fixed size units to devices such as disk drives

**Character Special file:** This type of file providing unbuffered I/O access in variable sized units to devices. All devices on a system are wither block special files ot character special files.



FIFO: Type of file used for communication between processes. Sometimes called as named pipe.

Socket: A type of file used for network communication between processes. A socket also be used for non network communication between processes on a single host. We use sockets for inter process communication.

Symbolic link: A type of file that points to another file. This file is encoded in st\_mode member of the stat structure.

We can determine the type of file in the macros shown in given table

Macro	Type of file
S_ISREG()	Regular file
S_ISDIR()	Directory file
S_ISCHR()	Character special file
S_ISBLK()	Block special file
S_ISFIFO()	Pipe or FIFO
S_ISLNK()	Symbolic link
S_ISSOCK()	Socket

## File access Permissions: st\_mode value enables access permission bits for the file

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



## Ownership of New files and directories

When user open a file kernel performs its access tests based on the effective user and group ids. When process will be run by someone else using `set_user)id` and `set_group_id` then this function is useful

The access function tests on the real user and group ids.

```
int access(const char* pathname, int mode);
```

Returns 0 if OK

01 on error

For this function we have to include

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

mode	Description
R_OK	Test to read permission
W_OK	Test for write permission
X_OK	Test for execute permission
F_OK	Test for existence of file



## Example F\_OK flag

```
#include<sys/stat.h>
#include<fcntl.h>
extern int errno;
int main(){
int fd = access("sample.txt", F_OK);
if(fd == -1){
printf("Error Number : %d\n", errno);
perror("Error Description:");
}
else
printf("file is accessed No error\n");
return 0;
}
```

Check for  
all  
permission  
bits (read,  
write,  
execute)

```
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
extern int errno;
int main(){
int fd = access("sample.txt", (R_OK | W_OK)&&
X_OK);
if(fd == -1){
printf("Error Number : %d\n", errno);
perror("Error Description:");
}
else
printf("file is accessed No error\n");
return 0;
}
```

## Unmask function:

Mask	Bit meaning
0400	User read
0200	User-write
0100	User-execute
0040	Group-read
0020	Group-write
0010	Group-execute
0004	Other-read
0002	Other-write
0001	Other-execute

The `unmask()` system call sets the file mode creation mask for the process and returns the previous value.

Syntax:

```
mode_t unmask(mode_t cmask)
```

Returns previous file mode creation mask.

## Chmod and fchmod functions

These two functions allow user to change the file access permissions for an existing file.

`Int chmod(const char*pathname, mode_t mode)`

`Int fchmod(int filedes, mode_t mode)`

Both function return 0 if OK

Or return -1 for error

The chmod function operates on the specified file where as fchmod function operates on a file that has already been opened.

To change the permission bits of a file the effective user ID or the process must be equal to the owner ID of the file/ process may have superuser permissions

[chmod\(2\) - Linux manual page \(man7.org\)](http://man7.org)

## Sticky bit

S\_ISVTX: : this bit was known as sticky bit as it is used in demand paging.

The text portion of the file stuck around in the swap area until the system was rebooted. Sticky bit is to be set for directory.

If the bit is set for directory, then then file can be removed or renamed only if the user has write permission for the directory and one of the following:

- 1) Owns the file
- 2) Owns the directory
- 3) Is the super user
- 4) The directories/tmp and /var/spool/ucpublic are typical candidates for the sticky bit. User can do with his own created files not with files created by another user.

## Chown, fchown and lchown functions

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

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

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

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

Returns 0 if ok

Returns -1 if error

These functions allow us to change the userID of a file and the groupID of a file.

These three functions operate similarly unless the referenced file is a symbolic link.

File size: the `st_size` member of the `stat` structure contains the size of the file in bytes.  
For regular file size: 0 is allowed.  
For directory: it will be some number  
For symbolic link: number of bytes in the filename / path of the file  
e.g. `/usr/lib` means size is 7



## File Truncation

A file can be truncated by copying of data at the end of the file.

Emptying a file which can do with O\_TRUNC flag to open ( This is special case of truncation)

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

```
int truncate(const char * pathname, off_t length)
```

```
int ftruncate(int fildes, off_t length)
```

both return 0 on success and -1 on error

These two functions truncate an existing file to length bytes.

The ftruncate function is used to empty a file after obtaining a lock on file.

# FILE SYSTEMS



**Link,  
unlink,  
remove  
and  
rename  
functions**



**Symbolic  
links**



**Symlink  
and  
readlink  
functions**



**File  
times**



**Utime  
function**



**mkdir,  
rmdir  
functions**



**Reading  
directories**

## Link system call

```
int link(const char*existingpath, const char *newpath)
```

Returns 0 if OK otherwise error

link() creates a new link (also known as a hard link) to an existing file.

If newpath exists, it will not be overwritten. This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the "original".

## deadlock scenario for link() system call

Set of processes already in deadlock state, every process in the set is waiting for another process for resource then another process can release. This will go in continuous loop so manual intervention is needed to overcome from deadlock State.

One resource should be kept in non-sharable mode and Process may for another process to acquire additional resource. Resources cannot be pre-empted so circular wait exists between processes. Different protocols are used to ensure deadlock cannot happen or to overcome from this situation. Deadlock prevention or detection and avoidance.

## Unlink system call

To remove existing directory entry we call the unlink Function.

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

It returns 0 if ok otherwise error

This function removed directory entry and decrements the link count of the file referenced by pathname. To unlink a file we must have permission and execute permission in the directory containing the directory entry.

When the link count reaches to 0 file will be deleted.

When file is open it will not be deleted.

When file is closed kernel check whether any other process opened the file if not then it will also check link count if it is 0 then files contents are deleted.

remove

```
#include<stdio.h>  
int remove(const char * pathname)  
Return 0 if ok -1 on error
```

rename

```
#include<stdio.h>  
int rename(const char * oldname, const char*newname)  
Return 0 if ok -1 on error
```

## Symbolic Link

A symbolic link is an indirect pointer to a file. Which pointed directly to the i-node of the file. These links were introduced to get around the limitations of hard links such as:

- i) Hard links normally require that the link and the file reside on the same file system.
- ii) Only the superuser can create a hard link to a directory. Anyone can create symbolic link to a directory. They are used to move file or an entire directory hierarchy to another location on a system.

## Symlink and readlink functions

A symbolic link is created with symlink function.

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

```
int symlink(const char* actualpath, const char * sympath)
```

Return 0 if OK -1 on error

A new directory entry sympathy is created that pint to actual path.

Actual path and sympathy need not reside in the same file system.

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

```
ssize_t readlink(const char * restrict pathname, char * restrict buf, size_t  
buffsize)
```

Return: number of bytes read if ok -1on error

The readlink function combines the actions to open ,read and close. If the function is successful it returns the number of byes placed into buf.

The content of the symbolic link that are returns in buf are not null terminated.



## File times

### File Times:

Three time fields are maintained for each file as follows:

Field	Description	Example	Ls(1) Option
st_atime	Last access time of file data	Read	-u
st_mtime	Last modification time of file data	Write	default
st_ctime	Last change time of i-node status	chmod, chown	-c

st_atime	Last access time of file data	Read -u
st_mtime	Last modification time of file data	Write default
st_ctime	Last change time of i-node status	chmod, chown-c

The difference between the modification time(st\_mtime) and changed status time(st\_ctime) is

mtime: when the contents of the file were last modified

ctime when the inode of the file was last modified.( such as changing file access permissions,changing user ID< changing number of links)

## File times

utime function

```
#include<utime.h>
```

```
int utime(const char * pathname, const struct utimbuf  
*times)
```

Return 0 if ok -1 on error

Structure used is

```
struct utimbuf
```

```
{
```

```
time_t actime;//access time
```

```
time_t modtime;//modification time
```

```
}
```

These are calendar times counts time in seconds (with epoch)

The operation of this function and the privileges required to execute it depend on whether the time argument is NULL.



## File times

### utime function

If times are NULL pointer the access time and modification time are both set to the current time.

If times are non null pointer the access time and the modification time are set to the value sin the structure pointer to by times.

## mkdir and rmdir functions

### mkdir function

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

```
int mkdir(const char *pathname, mode_t mode)
```

Return 0 if ok -1 on error

This function created new empty directory. The entries for . And .. Are automatically created

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

```
int rmdir(const char * pathname)
```

Return 0 if OK -1 on error

If the link count of the directory becomes 0 and no other process have opened directory then this function frees the space occupied by this directory. If one or more processes have the directory open when the link count reaches to 0 the last link is removed and the dot and dotdot entries are removed before this function returns.

## Reading directories

Directories can be read by anyone who has access permission to read the directory but only the kernel can write to a directory to preserve file system sanity.

For this following functions are used.

chdir, fchdir and getcwd functions

1)chdir function

2)fchdir function

3)getcwd function

## Reading directories

chdir function;

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

```
int chdir(const char * pathname)
```

```
int fchdir(int filedes)
```

Both functions return 0 if ok otherwise return -1 on error

We can specify the new current working directory either as a pathname or through an open file descriptor.

example

```
if(chdir("/tmp")<0)
```

```
err_sys("chdir failed")
```

```
printf("directory is successfully changed");
```

```
exit(0);
```

The task of this function that starts at the current working directory dot and worked its way up the directory entry hierarchy using dot-dot to move up one level.

## getcwd function

getcwd function gets the current working directory

Syntax

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

Returns buf if ok otherwise an error

We have to pass address as buf to this function and its size in bytes.

The getcwd is useful function provides when we have an application that needs to return to the location in the file system where it started out.

## Advanced file I/O

Linux provides a list of advanced I/O system call such as:  
**Scatter/gather I/O:** It allows single call which reads /write data from many buffers and it also helps on group together the fields of different data structures. This is also called as a **Vectored I/O**.

Advantages over Linear I/O:

- 1) **Efficiency:** a single vectored I/O can replace a number of linear I/O operations.
- 2) **Performance:** Internal optimization improves the performance.
- 3) **Atomicity:** A process can execute a single vectored operation with no risk of interleaving.
- 4) **Method of handling:** It helps to handle data more naturally. A set of segments is called a vector where each segment in the vector constitutes the address and length of the buffer in memory to or from where the data should be read or written.



```
struct iovec
{ void
*iov_base;
Size_t
iov_len;
};
```

The function( system calls) used in vector I/O are as follows:  
readv(): this function reads count segments from the file descriptor fd into the buffers described by iov(input/output vector)

Syntax:

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

```
ssize_t readv(int fd, const struct iovec *iov, int count);
```

iovec is a segment.

writev: This function writes at most count segments from the buffers described into the file descriptor fd.

```
ssize_t write(int fd, const struct iovec *iov, int count);
```

On successful execution both the function will return number of bytes read or written.

## Mapping files into memory

prot  
parameter  
describes  
desired  
memory  
protection  
of the  
mapping.

```
void *mmap(void *addr, size_t len, int prot, int flags,  
int fd, off_t offset)
```

fd → file descriptor

offset → starting offset bytes

len → length bytes of the object

Prot--: access permissions

flags → additional behaviour

Addr → a preference to use that starting address in the memory.

The kernel provides an interface which allows an application to map a file into memory, This can be done using mmap() system call for mapping objects into memory and a file is accessed directly through memory.

PROT\_READ: The pages may be read

PROT\_WRITE: The pages may be written

PROT\_EXEC: The pages may be executed

## Advantages of mmap()

- i) The mmap() function helps to avoid a copy of data while read() and write() system calls
- ii) The memory mapped files are accessed in a simple manner.
- iii) No need of lseek() system call

## Disadvantages of mmap()

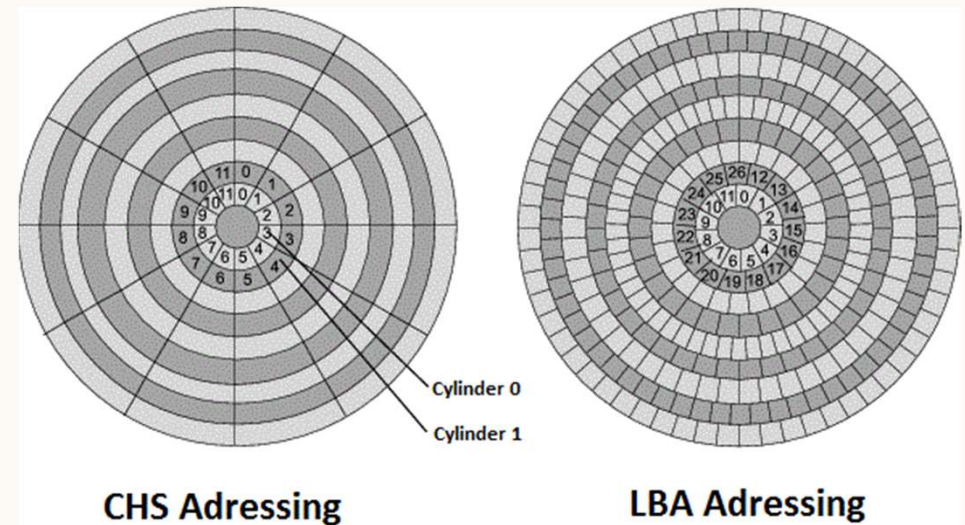
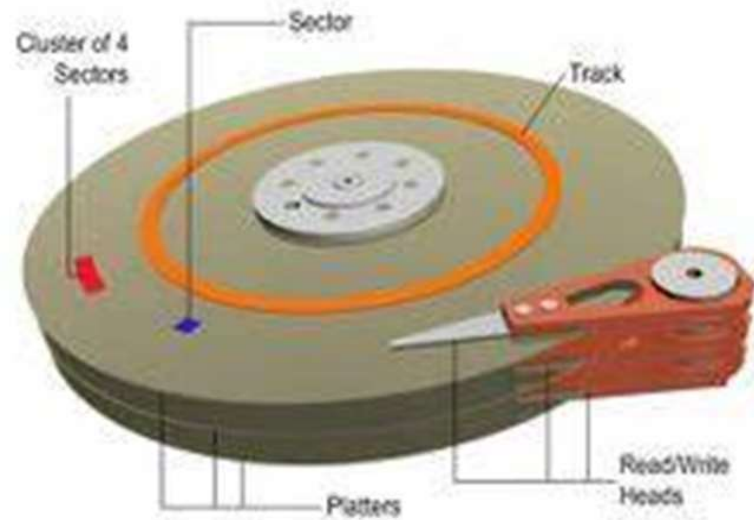
- i) There is an overhead in creating and maintaining the memory mappings associated data structures inside the kernel
- ii) Memory mapping is done by taking into consideration the number of pages in size which wastes the memory if the file is small
- iii) The memory mappings must be done into processes address space.

## I/O schedulers

Kernel implement the I/O schedulers which minimize the number and size of disk seeks by manipulating the order in which I/O requests are serviced and the times at which they are serviced.

- i) Disk Addressing:
- ii) The life of an I/O scheduler:
- iii) Optimizing I/O performance:

## i) Disk Addressing:



To locate specific unit of data on a disk, the drives logic requires three information components such as cylinder, head and sector values.

**The modern disk drives map a unique block number over each cylinder/head/sector. Then the disk is addressed using these block numbers. This process is called LBA addressing.**  
**The hard drive internally translates the block number into the correct CHS cylinder/head/ sector address.**

The life of an I/O scheduler: Two basic operations are performed by an I/O scheduler namely

**Merging and sorting**

**Merging:** It combines two or more adjacent I/O requests into a single request

**Sorting:** It arranges the I/O operations in the increasing block order

**Because of above two operations the disks head's movements are minimized.**

Optimizing I/O Performance: I/O performance can be optimized by considering the following factors such as:

- i) **Using user buffering**
- ii) **Performing block size aligned I/O**
- iii) **Using vectored I/O**
- iv) **Asynchronous I/ O**
- v) **Taking advantage of advanced I/O techniques.**

# Files and their metadata

An inode is a physical object located on the disk of a file system and an entity represented by a data structure in the linux kernel.

The inode stores metadata associated with a file such as files access permission, last access timestamp owner, group size as well as location of the files data. The inode can be obtained using command:

```
$ ls -li
```

The stat family function line stat, fstat and lstat returns information about file.

# Files and their metadata

```
#include<sys/types.h>
#include<sys/stat.h>
#include<unistd.h>
int stat(const char * path, struct stat *buf)
int fstat(int fd, struct stat *buf)
int lstat(const char * path, struct stat *buf)
```



# Copying and moving files

Unix provide a system call for moving files as:

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

```
int rename (const char * oldpath, const char* newpath)
```

A successful call to rename function renames the pathname oldpath to new path. The files content and the inode remains the same.

Steps for copying a file a.txt to a file named d.txt are as follows:

- 1) Open a.txt( source file name)
- 2) Open d.txt(destination file name), creating it if it does not exist and truncating it to zero length if it exists
- 3) Read a chunk of a.txt in memory
- 4) Write the chunk to d.txt
- 5) Continue till all the a.txt has been read and written to d.txt
- 6) Close d.txt
- 7) Close a.txt

For directory copy the individual directory and any subdirectories can be created using mkdir system call.

# THANK YOU

[GitHub - suvratapte/Maurice-Bach-Notes: Notes on the classic book: The Design of the UNIX Operating System](#)