Contents

[Linux file-system related commands : 2](#_Toc511466160)

[System calls for File-systems 3](#_Toc511466161)

[File System – functions 4](#_Toc511466162)

[Synchronous/asynchronous read/write operations 9](#_Toc511466163)

[Hard link vs Soft link 9](#_Toc511466164)

[Buffer cache vs cache memory 9](#_Toc511466165)

[File system 14](#_Toc511466166)

[Page size vs Block size 14](#_Toc511466167)

[INODES 14](#_Toc511466168)

[File structure in UNIX 16](#_Toc511466169)

[Superblock’s free inode list 16](#_Toc511466170)

[Superblock’s free disk block list 16](#_Toc511466171)

[Doubts 16](#_Toc511466172)

[1Buffer cache => header contents 10](#_Toc511236372)

[2Location of buffer cache 11](#_Toc511236373)

[3Why to use buffer cache 12](#_Toc511236374)

[4INODE structure in UNIX 16](#_Toc511236375)

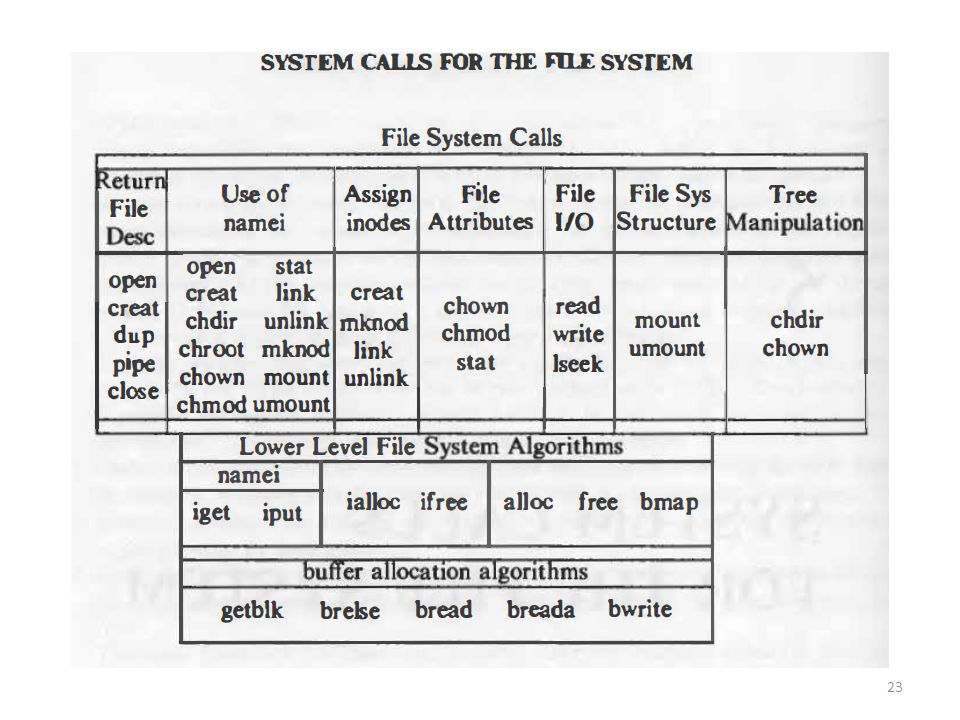
1File system imp information/links

|  |
| --- |
| lseek => this command doesn’t access the inode structure of the file since we are not accessing any data present inside the file, but only the specific process’s file descriptior table is accessed for knowing the sizeof the file and its current offset. |
| When a process terminates all of its open files are closed automatically by the kernel. |
| Buffer cache is a software structure(for file system) also buffer cache is present on main memory , cache memory is a hardware that speed memory references |
| Buffer cache => buffer header (contains logical device number not the physical device number, block no, status, ptr to data area, ptr to previous buf on has queue, ptr ot next buf on has queue,ptr to previous buf on free list, ptr to next buf on free list etc..), buffer data |
| <http://edusagar.com/articles/view/23/Inode-file-structure-on-Unix>  <http://edusagar.com/articles/view/24/inode-and-file-descriptor-table-Interaction>  <https://www.youtube.com/watch?v=IFGxQGuLNbs>  https://stackoverflow.com/questions/5315428/how-to-create-a-file-with-file-holes  <https://lwn.net/Articles/440255/>  <https://www.youtube.com/watch?v=0Q1oFPC1ElI&list=PLLDC70psjvq5hIT0kfr1sirNuees0NIbG&index=28> → VVIMP  <http://slideplayer.com/slide/9087341/> → VVIMP  <http://fwheel.net/aio.html> → VVIMP  <https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch04s05.html> → VVIMP  http://osr507doc.xinuos.com/en/PERFORM/mp\_nhbuf.html→ controlling no of buffers vs hash queue mod no |
|  |
|  |
|  |
|  |

## Linux file-system related commands :

|  |
| --- |
| * + cat /proc/fd |
| * + od –c file\_name => prints the file contents as characters |
| * + dd command |
| * + cat /proc/meminfo |
| * + free –m |
| * + df –h |
| * + sync, fsync, fdatasync |
| * + od –c file\_name |
| * + dd |
| * + cat /dev/fd |
| * + mount |
| * + mkfs |
| * + ln –s => to create symbolic link |
| * + /etc/fstab |
| * + ls command – list directory contents.   + du command – estimate file space usage.   + stat command – display file or file system status. |

# System calls for File-systems



# File System – functions

* **#include <fcntl.h>**

int fd = open(const char\*filename, int flag, …., mode\_t mode);

* Any process by default will have 3 file descriptors : 0 – STDIN, 1-STDOOUT, 2-STDError, now if process tried to open a new file, its might probably get its new fd as **4**

2open function parameters=> types

|  |
| --- |
| O\_RDONLY  O\_WRONLY  O\_RDWR  O\_APPEND  O\_TRUNC : if the file exists and it is successfully opened for either readwrite or write-only ,truncate its length to 0  O\_CREAT : if the file is not present a new file is created, else if the file already exists it just opens , This option requires a third argument to the open function, the *mode*, which specifies the access permission bits of the new file.  O\_EXCL : *If O\_CREAT and O\_EXCL are set, open() shall fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist shall be atomic with respect to other threads executing open() naming the same filename in the same directory with O\_EXCL and O\_CREAT set. If O\_EXCL and O\_CREAT are set, and path names a symbolic link,open() shall fail and set errno to [EEXIST], regardless of the contents of the symbolic link. If O\_EXCL is set and O\_CREAT is not set, the result is undefined.*  O\_NONBLOCK  O\_SYNC : each write wait for physical I/O to complete and also including IO necessary to update file attributes modified as a result of write. **These 2 operations happen in atomic mode**  O\_DSYNC : each write wait for physical I/O to complete **but don’t wait for file attributes to be updated** if they don’t affect the ability to read the data just written.  O\_RSYNC : Have each read operation on the file descriptor wait until any pending writes for  the same portion of the file are complete. |

* New file creation :
  + int fd = open( “/tmp/a.c”, O\_WRONLY|O\_CREAT|O\_TRUNC, mode);
  + int fd = creat(“/tmp/a.c”, mode); When a file is created using **creat** function, file is opened for only writing, if we want to read frm the file then again we have to do => close the opened file , open file with O\_RDWR and then perform read operations.
* **Hole in a file**

* + **<https://stackoverflow.com/questions/5315428/how-to-create-a-file-with-file-holes>**
  + TO create a file with a hole :
    - Create a new file, seek to position N , write some data ( now in this case from 0 till offset-N there is no data present), so a hole has been created.
  + When a hole is created in a file, that hole if we read it using **read** function data is read back as **0**. Also this hole is not stored on the disk, **so file size doesn’t change even though there are holes inside a given file.**
* When a process terminates all of its open files are closed automatically by the kernel.
  + int close(int fd);
* lseek :
  + <https://lwn.net/Articles/440255/>
  + By default, this offset is initialized to 0 when a file is opened, unless the O\_APPEND option is specified.
  + #include <unistd.h>

off\_t lseek(int fd, off\_t offset, int whence)

* + Return value of lseek function => returns the new file offset.
  + ***lseek****() allows the file offset to be set beyond the end of the file (but this does not change the size of the file). If data is later written at this point, subsequent reads of the data in the gap (a “hole”) return null bytes (‘\0’) until data is actually written into the gap.*
  + **Lseek operation doesn’t access INODE structure since in lseek only “fd offset” position is changed which is present in Process PCB ==> struct task\_struct => fd number => struct file**
  + In order to get the current position in the file , we can seek zero bytes from the current position to determine the current offset

int current\_position = lseek( fd, 0, SEEK\_CUR);

3lseek => whence options

|  |
| --- |
| * **SEEK\_SET** :  The file offset is set to offset bytes from the beginning of the file * **SEEK\_CUR** : The file offset is set to current location + offset * **SEEK\_END** : The file offset is set to size of file + offset bytes. * If whence is SEEK\_HOLE, the offset of the start of the next hole greater than or equal to the supplied offset is returned. * If whence is SEEK\_DATA, the file pointer is set to the start of the next non-hole file region greater than or equal to the supplied offset.   A “hole” is defined as a contiguous range of bytes in a file, all having the value of zero, but not all zeros in a file are guaranteed to be represented as holes returned with SEEK\_HOLE. Filesystems are allowed to expose ranges of zeros with SEEK\_HOLE, but not required to. Applications can use SEEK\_HOLE to optimize their behavior for ranges of zeros, but must not depend on it to find all such ranges in a file. The existence of a hole at the end of every data region allows for easy programming and implies that a virtual hole exists at the end of the file. **Applications should use fpathconf(\_PC\_MIN\_HOLE\_SIZE) or pathconf(\_PC\_MIN\_HOLE\_SIZE) to determine if a filesystem supports SEEK\_HOLE.  For filesystems that do not supply information about holes, the file will be represented as one entire data region**. |

* Size of a file with holes and without holes will be the same but if we check the files using **od** command then we observe that the file without holes occupies more number of disk blocks and the file with holes occupies less number of disk blocks.
* Another way to create holes In a file is using **dd** command
* Read :
  + int read(int fd, char \*buf, int no\_of\_bytes);
  + returns no of bytes read on success, 1 on error , 0 if EOF
* Write
  + int write(int fd, const char \*buf, int no\_of\_bytes);
  + For a regular file, the write starts at the file's current offset. If the O\_APPEND option was specified when the file was opened, the file's offset is set to the current end of file before each write operation. After a successful write, the file's offset is incremented by the number of bytes actually written.
* Linux command : **od** : can be used to dump the file contents in hex format and also can be used to check the holes in the file
* Following set of operations are not atomic :



* + **#include <unistd.h>**
  + **ssize\_t pread(int *fd*, void \**buf*, size\_t *count*, off\_t *offset*);**

**pread() reads up to *count* bytes from file descriptor *fd* at offset *offset* (from the start of the file) into the buffer starting at *buf*.**

**The file offset is not changed.**

* + **ssize\_t pwrite(int *fd*, const void \**buf*, size\_t *count*, off\_t *offset*);**

**pwrite() writes up to *count* bytes from the buffer starting at *buf* to the file descriptor *fd* at offset *offset***

**The file offset is not changed**.

* Duplicating fd
  + #include <unistd.h>
  + int dup(int filedes );
    - The new file descriptor returned by dup is guaranteed to be the lowest-numbered available file descriptor.
  + int dup2(int filedes, int filedes2);
    - With dup2, we specify the value of the new descriptor with the *filedes2* argument. If *filedes2* is already open, it is first closed. If *filedes* equals *filedes2*, then dup2 returns *filedes2* without closing it.
  + Example :







* sync/fsync/fsyncdata =>returns 0 on success and 1 on error
  + When we write data to a file, the data is normally copied into the kernel into one of its buffers(i.e.buffer cache) and queued for writing to disk at sometime later this is called as **DELAYED WRITE. Kernel** eventually writes all the delayed-write blocks at a later point of time,normally when it needs to reuse the buffer at a later point of time
  + **#include <unistd.h>**
  + void sync(void) => **this is asynchronous operation**
    - The sync function **simply queues all the modified block buffers for writing and returns; it does not wait for the disk writes to take place.** The function sync is normally called periodically (usually every 30 seconds) from a system daemon, often called update. This guarantees regular flushing of the kernel's block buffers.
  + int fsync(int *filedes*); => **this is synchronous operation both file attributes+data portions of the file**
    - **The function fsync refers only to a single file, specified by the file descriptor *filedes*, and waits for the disk writes to complete before returning.** With fsync, the file's attributes are also updated synchronously. The intended use of fsync is for an application, such as a database, that needs to be sure that the modified blocks have been written to the disk.
  + int fdatasync(int *filedes*);
    - The fdatasync function is similar to fsync, but it affects only the data portions of a file.

## Synchronous/asynchronous read/write operations

* <http://fwheel.net/aio.html>
* https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch04s05.html

## Hard link vs Soft link

## Buffer cache vs cache memory

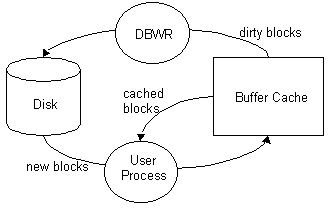
* <https://www.youtube.com/watch?v=x2vegjeJICk>
* **<https://www.tldp.org/LDP/sag/html/buffer-cache.html>**
* **http://osr507doc.xinuos.com/en/PERFORM/mp\_nhbuf.html**

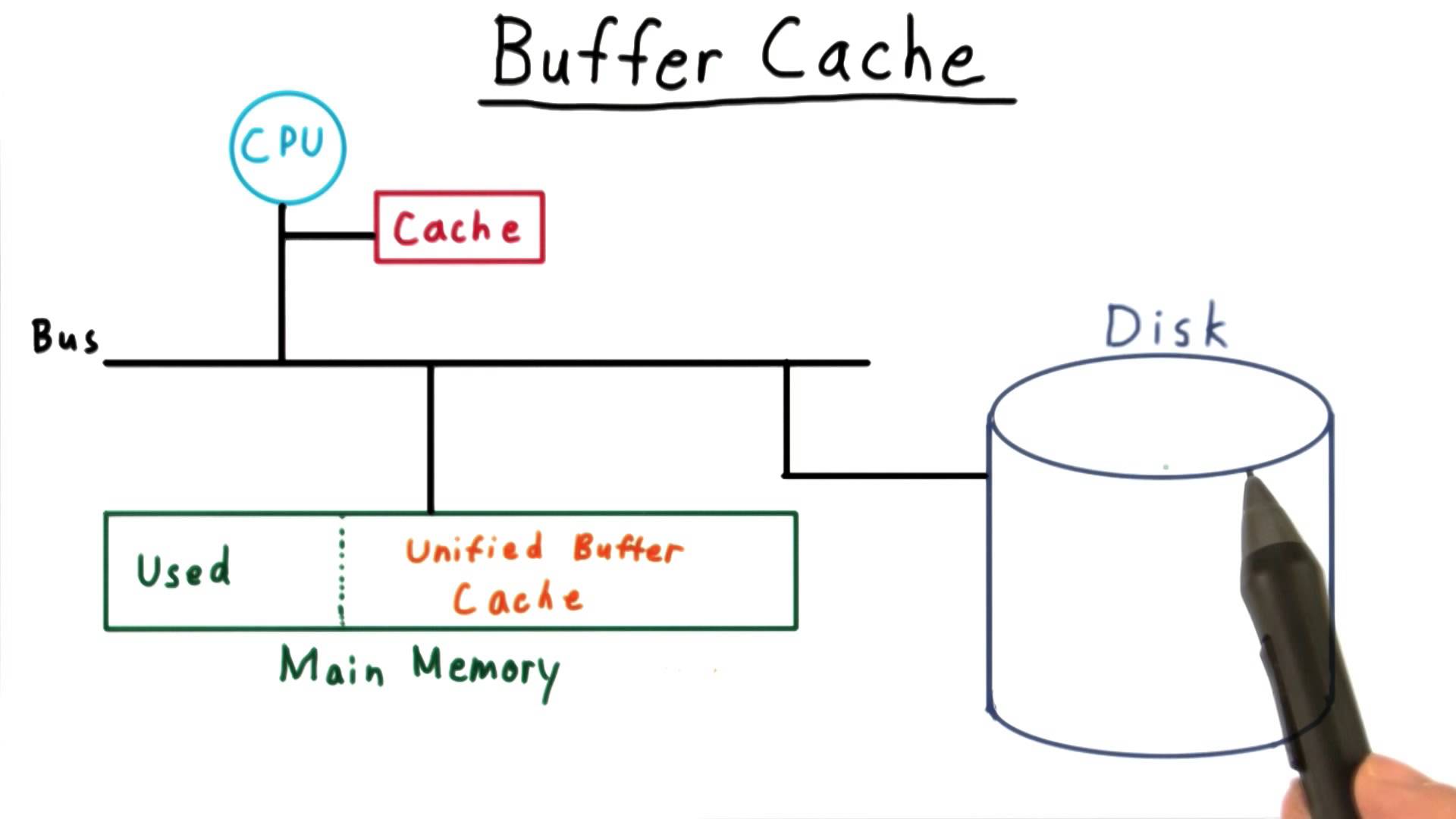
|  |
| --- |
| * <https://www>.slideshare.net/vijaychandraker/unix-ch03032 * <http://unixbyrahul>.50webs.com/unix3.html * <http://www>.science.unitn.it/~fiorella/guidelinux/tlk/node112.html * <https://sqljunkieshare>.com/2012/03/15/how-does-the-buffer-manager-writes-the-data-in-the-sql-server/ |

* During system-operating system boot up process, buffer chace is created inside main memory.
* Buffer cache is present on Main memory, where s cache memory is a separate hardware block.
* The kernel while performing file read/writes attempts to minimize the frequency of disk access by keeping a pool of internal data buffers called buffer cache.
* When the system boots all the buffers are present on the free-buffer list.
* Buffer cache has 2 different queues:
  1. **Buffer hash queue : hashed as a function of the device number(device number is not the physical device no but it’s the logical device no.) and the block number. circular double linked list**
  2. **Free-buffer list :** ***circular double linked list***
* Mapping of buffer cache data to disk-block data is 1-1 mapping, i.e.a disk block can never map into more than one buffer at a time.
* Buffer cache
  1. Buffer header: contains a device number filed and a block number field that specify the file system and the block number of the data on the disk and uniquely identify the buffer . The device number is the logical file system number.
  2. Buffer data : the actual data from the disk
* Buffer cache header has the following fields:
  1. Locked status
  2. Is there another process waiting for the buffer cache to be unlocked
     + ***While a process tries to access buffer cache, first it locks the buffer cache and then after completion of its work on buffer cache the lock is unlocked.***
  3. Write delayed flag
  4. Device number (logical device number)
  5. Pointer flags : ptr to data area
  6. Ptr to prev buf on hash queue
  7. Ptr to next buf on hash queue
  8. Ptr to prev buf on free-buff queue
  9. Ptr to next buf on free-buff queue

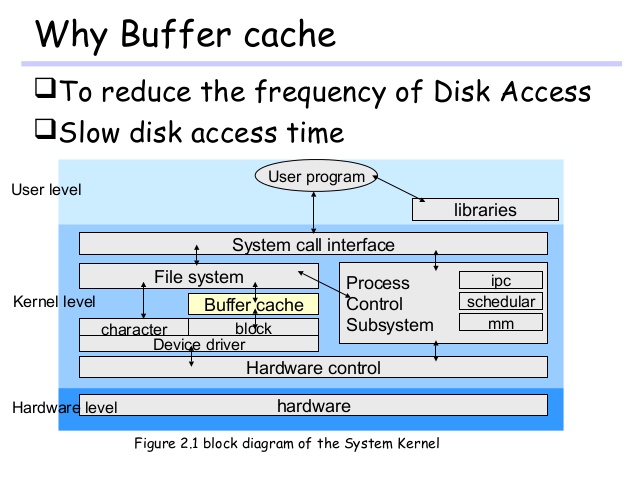


1Buffer cache => header contents

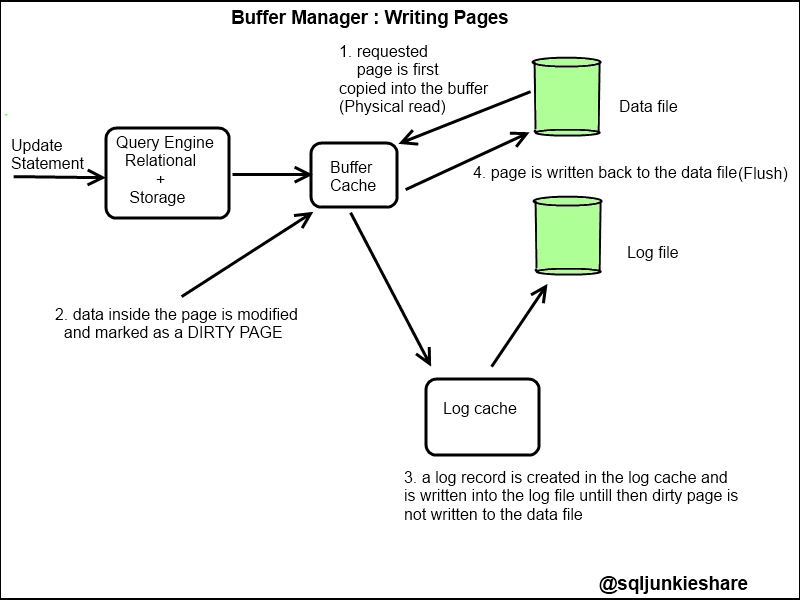


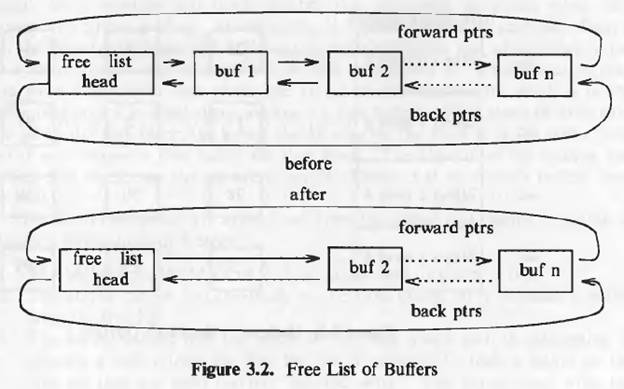


2Location of buffer cache



3Why to use buffer cache





* Buffer cache has two types of lists : hash queues and free buffer list
* A buffer maybe present on both hash queue and free buffer list at a given pointof time

## File system

* Linux generally counts on having twice the amount of physical memory in the form of swap space on the hard disk.
* During system startup, all the partitions are thus mounted, as described in the file /etc/fstab
* Use of the **su** (switch user) facility, which allows you to run a shell in the environment of another user, on the condition that you know the user's password.

## Page size vs Block size

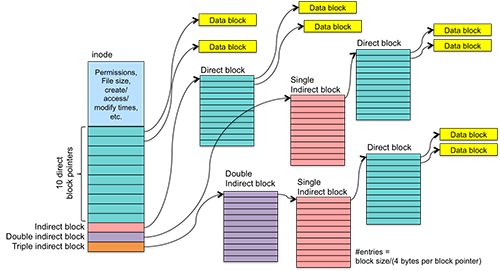
# INODES

* Inode contains information about a file and also where the data is stored on the hard-disk, bit INODE doesn’t contain actual file data.
* INODE present on hard-disk is also knows as DISK-INODE
* In main memory for inode (INCORE-INODE) there are two kind of buffers :
  + Hash queue => where inodes are stored
  + Free inode list => double circularly linked list.
* If a process wants to open/read/write to a file first the file’s inode is checked in inode- hash queue if it is not present then the inode is copied into main-memory which is then called as INCORE-INODE.
* INCORE-INODE has few fields in addition to DISK-INODE structure :

|  |
| --- |
| * + Locked => if process wants to read/write even a single byte of data process will lock the inode and get the block no. where the file data is present and then immediately process unlocks the inode   + Is there any other process waiting for the inode to be unlocked   + Flag indicating if INCORE-INDOE structure has been changed wrt INODE STRUCTURE   + Device number   + Block number   + Inode number : this parameter is not present in DISK-INODE because on disk all indoes are stored in a sequential manner, but while in case of INCORE-INODES they are not sequential   + **Reference count => if there are 2 process which has performed open functionality on the same file, then the reference count becomes 2, ONLY when a process does close(fd) the reference count decreases by 1 value.When the reference count becomes zero then it means that there is no process accessing this file and then the INCORE-INOD Eis moved to indoe-free-buffer list. For example : Process P1 wants to access /tmp/a.txt file’s data at offset 100 then P1**     1. **locks the inode**     2. **getblkAddr ( to know the block no where data corresponding to offset 100 is present)**     3. **unlocks the inode , but reference count is not changed**   + Original INODE information     1. **Type : file inode or directory inode**     2. **Status : allocated or unallocated** |

* **Only when the INCORE-INODE reference count becomes zero then inode is moved to INODE-free list which is present on main memory.**

### File structure in UNIX



4INODE structure in UNIX

### Superblock’s free inode list

* Whenever a process is trying to create a new directory or a new file, an inode which is present on the Superblock free-inode list is taken and given to the process

### Superblock’s free disk block list

## Doubts

* Inode
* Dentry
* File directory linux functions
* Soft vs hard link