DATA STORAGE TECHNOLOGIES & NETWORKS

(CS C446, CS F446 & IS C446)

LECTURE 28– STORAGE

Reading and Writing Disk Blocks

```
Algorithm bread
Input:     file system 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);
}
```

Algorithm for Reading a Disk Block

```
Algorithm breada
Input: (1) file system block number for immediate read
       (2) file system block number for asynchronous read
Output: buffer containing data for immediate read
{
    if(first block not in cache)
    {
        get buffer for first block (algorithm getblk);
        if (buffer data not valid)
            initiate disk read;
    }
    if(second block not in cache)
    {
        get buffer for second block (algorithm getblk);
        if (buffer data      valid)
            release buffer (algorithm brelse);
        else
            initiate disk read;
    }
    if (first read was originally in cache)
    {
        read first block (algorithm bread);
        return buffer;
    }
    sleep (event first contains valid data);
    return buffer;
}
```

Algorithm for Block Read Ahead

```
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;
}
```

Algorithm for Writing a Disk Block

UNIX file system

- Every file on a UNIX system has a unique id
  - It is known as inode number
- Inode contain information necessary for a process to access a file
  - File ownership, access rights, file size, and location of the file’s data in the file system.
- Kernel converts path + filename to the file’s inode.

Inode

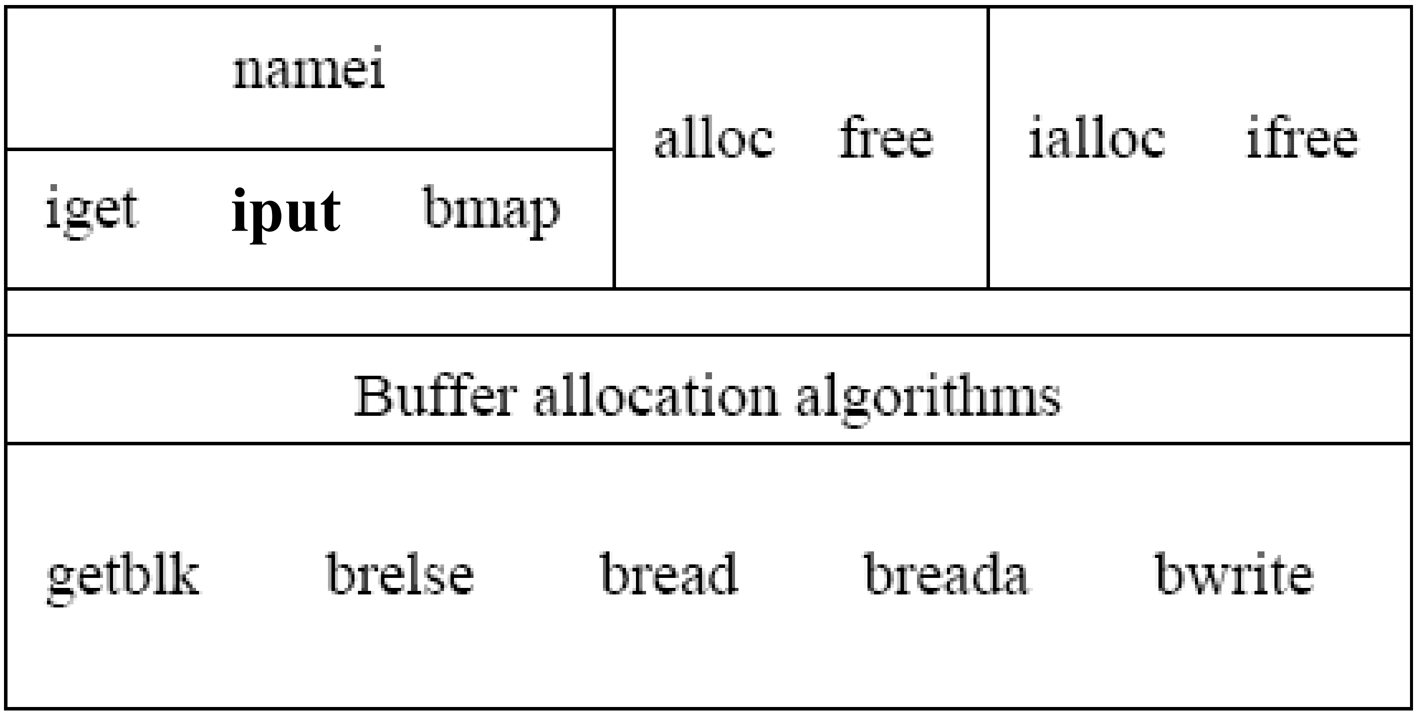
- Disk inode consist of following fields
  - File owner identifier
    - Individual, group, super user
  - File type
    - File, directory, special file or FIFO (pipes)
  - File access permissions
  - File access and modified times
  - Number of links to the file
  - Table of contents for the disk addresses of data in a file
  - File size, Block count

Inode does not specify the path name(s) that access the file

Changing the content of a file automatically implies a change to the inode, but changing the inode does not imply that the contents of the file change.

File System Algorithms

Lower Level File System Algorithms



File System Algorithms

Algorithm details

- iget → returns a previously identified inode, possibly reading it from disk via the buffer cache.
- iput → releases the inode
- bmap → sets kernel parameters for accessing a file
- namei → converts a user level path name to an inode, using the algorithm iget, iput and bmap.
  alloc → allocate disk blocks for files
- free → free disk blocks
- ialloc → assign inodes for file
- ifree → free inodes

In-core Inode

- In-core copy of the Inode contain following more information
  - The status of the in-core Inode, indicating whether
    - The Inode is locked
    - A process is waiting for the Inode to become unlocked
    - The in-core copy of inode differs from the disk copy as a result of a change to the data in the Inode
    - The in-core copy of Inode differs from the disk copy as a result of a change to the file data.
    - The file is a mount point.

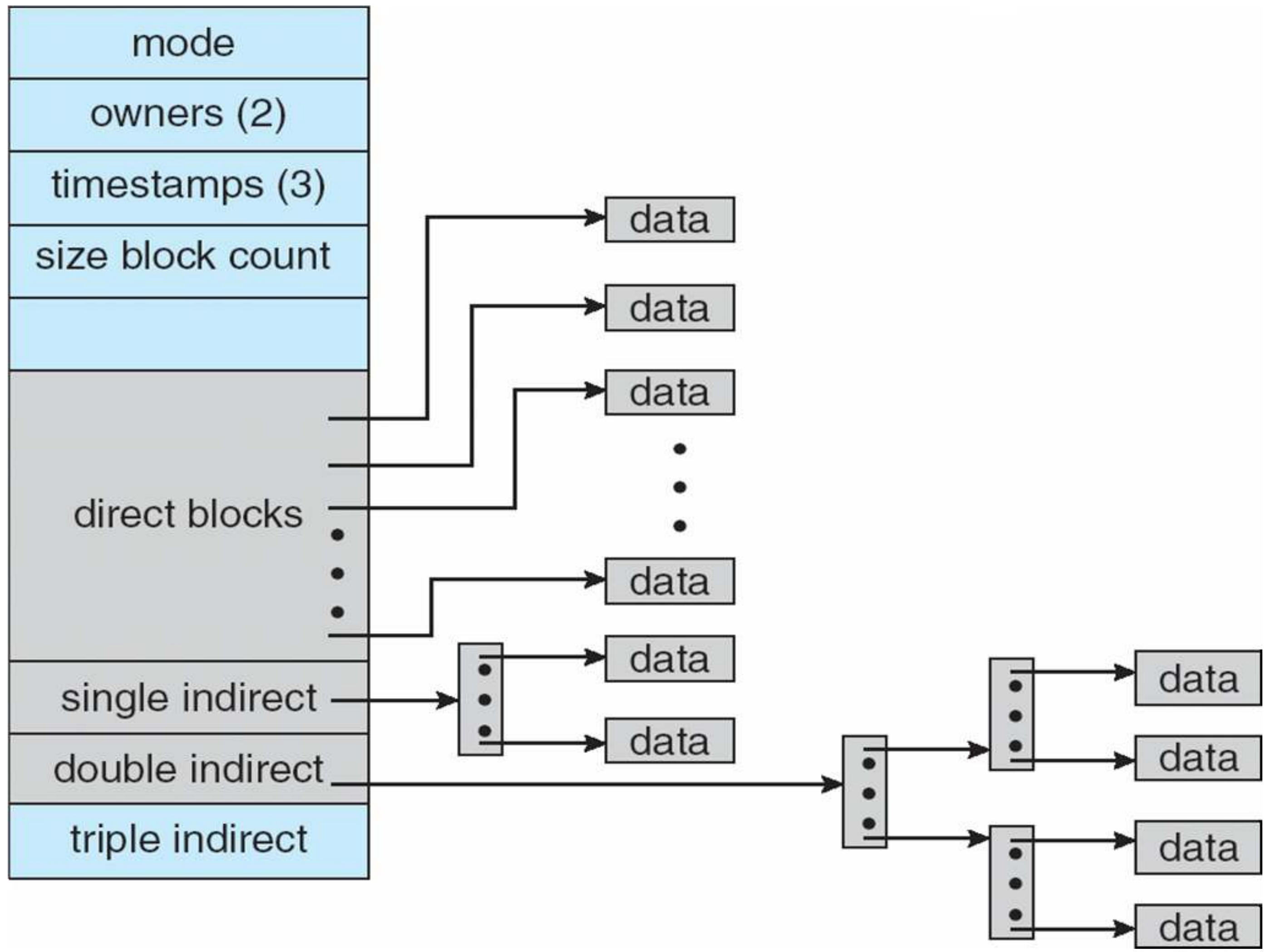
In-core Inode

- The logical device number of the file system that contains the file
- The inode number (inodes are stored in a linear array on disk)
- Pointers to other in-core inodes (kernel links inodes on hash queues
- A reference count, indicating the number of instances of the file that are active.
  - Inode will be in the free list only if the reference count is 0.

Inode structure

- 13 / 15 entries in inode table of contents
  - 10 / 12 direct
  - 1 single indirect
  - 1 double indirect
  - 1 triple indirect

Combined Scheme: UNIX (4K bytes per block)



bmap algorithm

- Processes access data in a file by byte offset.
- Kernel converts user view of bytes into view of blocks.
- File starts at logical block 0 and continues to a logical block number corresponding to the file size.
- Kernel accesses the inode and converts the logical file block into the appropriate disk block.
- bmap algorithm converts a file byte offset into a physical disk block.

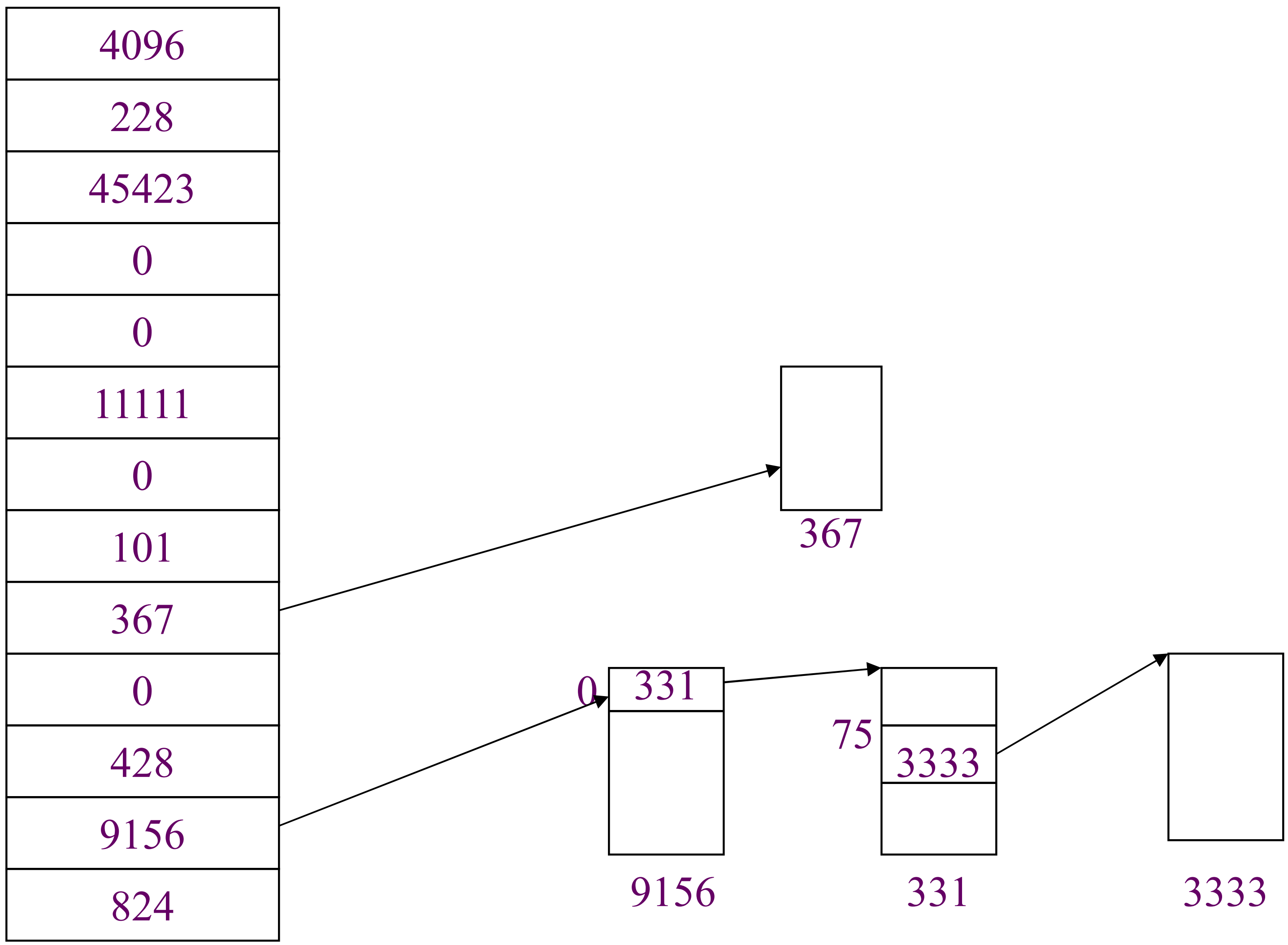
```
Algorithm bmap
Input:      inode, byte offset
Output:     block # in file system, byte offset into block, bytes of
            I/O in block, read ahead block #
{
            calculate logical block # in file from byte offset;
            calculate start byte in block for I/O;        /* output 2 */
            calculate # of bytes to copy to user;         /* output 3 */
            check if read ahead applicable, mark inode;/*output 4*/
            determine level of indirection;
            while (not at necessary level of indirection)
            {
                  calculate index into inode or indirect block from
                  logical block # in file;
                  get disk block # from inode or indirect block;
                  release buffer from previous disk read, if any
                  (algorithm brelse);
                  if (no more levels of indirection)
                        return block #;
                  read indirect disk block (bread algorithm);
                  adjust logical block # in file according to level
                  indirection;
            }
}
```

Conversion of Byte Offset to Block Number in File System

Example

- Assume
  - Logical block = 1 K Bytes
  - Block number is addressable by a 32 bit (4 bytes)
  - A block can hold 256 block numbers
  - Maximum number of bytes possible in a file = (10K + 256K + 64M +16G)bytes
  - Given that the file size field in the inode is 32 bits, the size of a file is effectively limited to 4GB (2^{32})

- Process access data in a file by byte offset
- The kernel converts the user view of bytes into a view of blocks
- File starts at logical block 0 and continues to a logical block number corresponding to the file size.
- The kernel accesses the inode and converts the logical file block number in to physical disk block.


4096
228
45423
0
0
11111
0
101
367
0
428
9156
824
367
0
331
75
3333
9156
331
3333

- E.g. 1
  - If a process wants to access byte offset 9000
    - Kernel calculate that the byte is in 8^{th} block (starting from 0)
    - It accesses the physical block number stored in 8^{th} direct map location. (If 367 is the value stored in 8^{th} direct map location then access 367^{th} block from the disk.)
    - Then access 808^{th} byte (9000 – 8192) in 367^{th} block.
    - This will be the resultant 9000^{th} byte in file.

- E. g. 2
  - If process want to access byte offset 350,000 in the file
  - 9th direct pointer can store till (10K-1)
  - Single indirect pointer can store from 10K to (256K+10K-1)
  - Double indirect pointer can store from 266K to (64M+266K-1)
  - The request is between these value. So the request is in double indirect pointer.

- 350000 in a file is 350000 – 266K in double indirection
- = 77616 of a double indirection.
- Since each single indirection can access 256K, 77616 is in the 0^{th} single indirection block of double indirection block.
- So we will fetch block number 331.
- Since the direct block in a single indirect block hold 1KB, 77616 is in 75^{th} direct block in the single indirect block.
- So we will fetch block number 3333.
- The offset location from where we need to access the byte is calculated by 77616 – 75K = 816
- If inode block entry is 0, then the logical block entries contain no data.