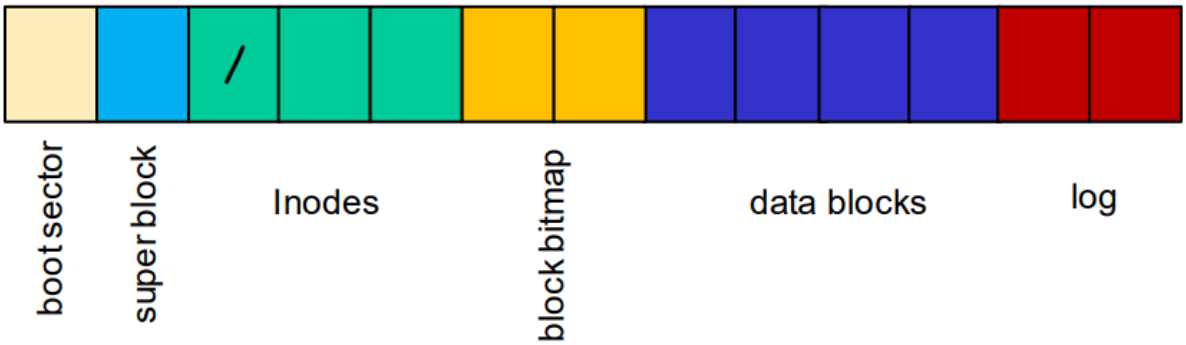
Extent File System:

On Disk File System Format:

- Both the User and kernel program uses this format.

Disk layout:

[boot block | super block | inode blocks | free bit map | data blocks | log]

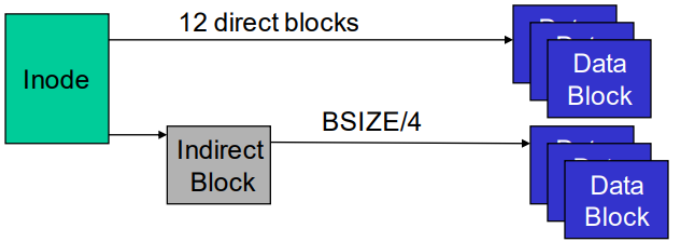


Super Block:

- It holds information about size of the file, number of data blocks allocated, number of inodes,log data's and starting address of these structure.
- Inorder to get some information about a file it checks the super bloc initially to get the details.
- Below is the data structure for the super block.

```
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
  uint bmapstart;    // Block number of first free map block
};
```

- In xv6 there are twelve direct pointers represented by NDIRECT = 12
- There is also one indirect pointer (512/4) = 128
- Therefore, there are 140 pointers.
- Initially each pointer points to a data block. But extent based file system can allocate continuous block of memory with just a single pointer. So it can support for large files.




```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE  (NDIRECT + NINDIRECT)
```

Dinode:

- Every file has an inode.
- The size of the inode is 64 bytes.
- Type =0 implies no file/directory
- Type=1 implies directory.
- Type=2 implies file.
- Nlink represent the no of files linked to this inode.
- It also has a size and an address attribute.
- Addrs is of 4 byte each so 52 bytes for 13 pointers.

```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses

};
```

File Data structure:

In memory structure:

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

- File data structure has type, reference count, inode pointer, offset.
- Like the ondisk inode the in memory data structure of the inode looks as shown below. In memory copy of the inode has all the fields like the on disk inode.

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];

};
```

Creation of O_EXTENT Flag:

- O_EXTENT flag is added to support opening of the new files of type extent in fcntl.h file.

```
#define O_RDONLY  0x000
#define O_WRONLY  0x001
#define O_RDWR    0x002
#define O_CREATE  0x200
#define O_EXTENT  0x004                    // added for pa4
```

- Inorder to open a file we use open() system call.

```
int
sys_open(void)
{
  char *path;
  int fd, omode;
  struct file *f;
  struct inode *ip;

  if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
    return -1;

  begin_op();
  if(omode & O_CREATE){
    if(omode & O_EXTENT){
      ip = create(path, T_EXTENT, 0, 0);
      if(ip == 0){
        end_op();
        return -1;
      }
    }
```

- The condition check for O_EXTENT is provided in the open() system call.

To display the contents of the EXTENT based file we have added T_EXTENT flag in stat.h file.

```
#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Special device
#define T_EXTENT 4 // Extend-based file
```

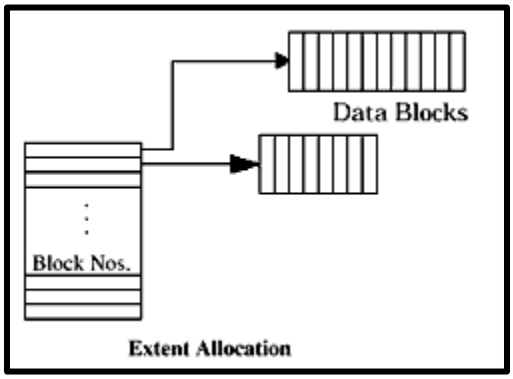
Data Structure of Stat system call:

- The below data structure is used to display the status of a file.

```
struct stat {
  short type;  // Type of file
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short nlink; // Number of links to file
  uint size;   // Size of file in bytes
};
```

Extent-based allocation

- Extent-based filesystems allocate disk blocks in large groups at a single time, which forces sequential allocation.
- As a file is written, a large number of blocks are allocated, after which writes can occur in large groups or clusters of sequential blocks.
- Filesystem metadata is written when the file is first created. Subsequent writes within the first allocation extent of blocks do not require additional metadata writes (until the next extent is allocated).
- This optimizes the disk seek pattern, and the grouping of block writes into clusters allows the filesystem to issue larger physical disk writes to the storage device, saving the overhead of many small SCSI transfers.
- a block address number is required for every logical block in a file on a block-allocated file, resulting in a lot of metadata for each file.



- So in extent based allocation each pointer holds the address and the number of data blocks that an address holds.
- First 3 bytes for the address and the next 1 byte for storing the length field.
- This is useful in for supporting large files.

Implementation of the Data Allocation is done in the Bmap() function:

```
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;
  if(ip->type==T_EXTENT){  // checking type of the file
    int k=0; // initializing variable k
    //if it already exist, e.g. for read operations
    for(k=0;k<(NDIRECT/2);k++){ // run through the length of the pointers here ndirect represent 12
      if(ip->addrs[k]==0) // if the address is zero
          break;
      if((ip->addrs[k+NDIRECT/2]+LAST_EIGHT_BITS(ip->addrs[k]))>bn && ip->addrs[k+NDIRECT/2]<=bn ) // this condition corresponds to the
neighbouring block check.
          return FIRST_THREE_BYTES(ip->addrs[k])+(bn- ip->addrs[k+NDIRECT/2]);
    }
    if(k==(NDIRECT/2)) // check for not allowing to allocate more extents for the file
      panic("bmap: You can't create more extents for the given file");

   uint blkadd= balloc(ip->dev); // allocate block using the device number as parameter.
   //the below condition is to increment the extents of the file.
   if(LAST_EIGHT_BITS(ip->addrs[k])<255 && (FIRST_THREE_BYTES(ip->addrs[k])+LAST_EIGHT_BITS(ip->addrs[k]))==blkadd)
      ip->addrs[k]+=1;
   // the below condition is useful to add new extents to the given file.
   else{

    ip->addrs[k]=COMBINE_ADD_TO_LEN(blkadd, 1); //address of 32bit combining first three bytes of block address and 1 byte of length as
address.
    ip->addrs[k+NDIRECT/2]=bn;
  }
   return blkadd; // return the address corresponding to the block.
```

- We start with a block. If there is a free neighboring block, then we just increment the length pointer.
- We proceed until we reach the neighboring block containing data.
- Once a neighboring block of address is encountered the we need to go to the next extent address. (ie) increment the inode pointer address and move to the next address.
- We continue to do so until we can support the max length of possible data block allocations.
- If the file system can’t support a very large file, then it displays bitmap out of range error.
- Each bit in the bit map is used to tell whether a data block is free, or it is in used.

Status of Extent File:

- **Status of an extent file is displayed using the fstat() system call.**
- **Inorder to display the status of an extent file Stat.c file is included.**
- **It involves opening of a file and calling fstat() system call using filepointer and address of the stat structure.**

```
int
print_stat(char *fileName){
  struct stat st;
  int fd;
        if((fd = open(fileName, 0)) < 0){
            printf(2, "ls: can't open %s\n", fileName);
        }
       if(fstat(fd, &st) < 0)
       {
          printf(1, "ls: can't stat %s\n", fileName);
          close(fd);
       }
  printf(1,"\nfile: %s\ninode number: %d\nsize: %d\ndev number: %d\n",fileName,st.ino,st.size,st.dev);
  switch(st.type){
    case T_EXTENT:
      printf(1,"type: T_EXTENT\n");
      break;
    case T_DIR:
      printf(1,"type: T_DIR\n");
      break;
    case T_FILE:
      printf(1,"type: T_FILE\n");
      break;
    case T_DEV:
      printf(1,"type: T_DEV\n");
      break;
    default :
      break;
  }
```

```
int
main(int argc, char *argv[]){
  if(argc != 2){
    printf(1,"give the stat pathname");
    exit();
  }
  print_stat(argv[1]);
  exit();
}
```