# Operating Systems Practice

File system

Eunji Lee

(ejlee@ssu.ac.kr)

# Reference

- “xv6: a simple, Unix-like teaching operating system,” Chapter 7. File system

# Challenges in File system

- On-disk data structures
- Crash recovery support
- Coordination of concurrent operations
- In-memory cache management

# Overview

- Disk Layer reads and writes blocks on virtio hard drive
- Buffer Cache Layer caches disk blocks and synchronizes access to them
- Logging Layer wraps updates as a transaction and ensures the transaction is updated atomically
- Inode Layer provides individual files, each represented as an inode with a unique –number and some blocks holding the file's data
- Directory Layer implements each directory
- Pathname Layer resolves hierarchical pathname with recursive looksup
- File Descriptor Layer abstracts many UNIX resources usin

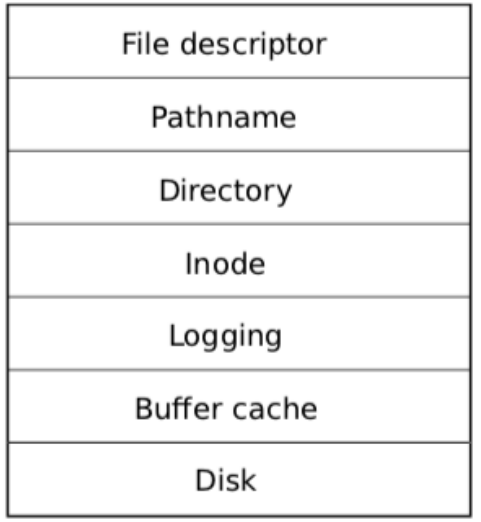



Figure 7.1: Layers of the xv6 file system.

# Overview

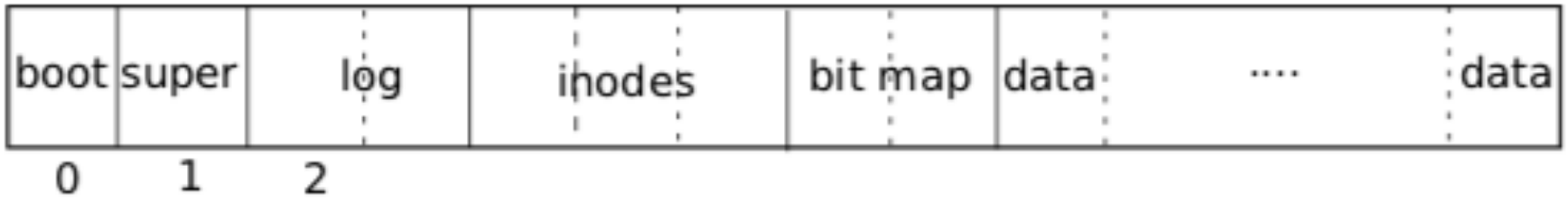
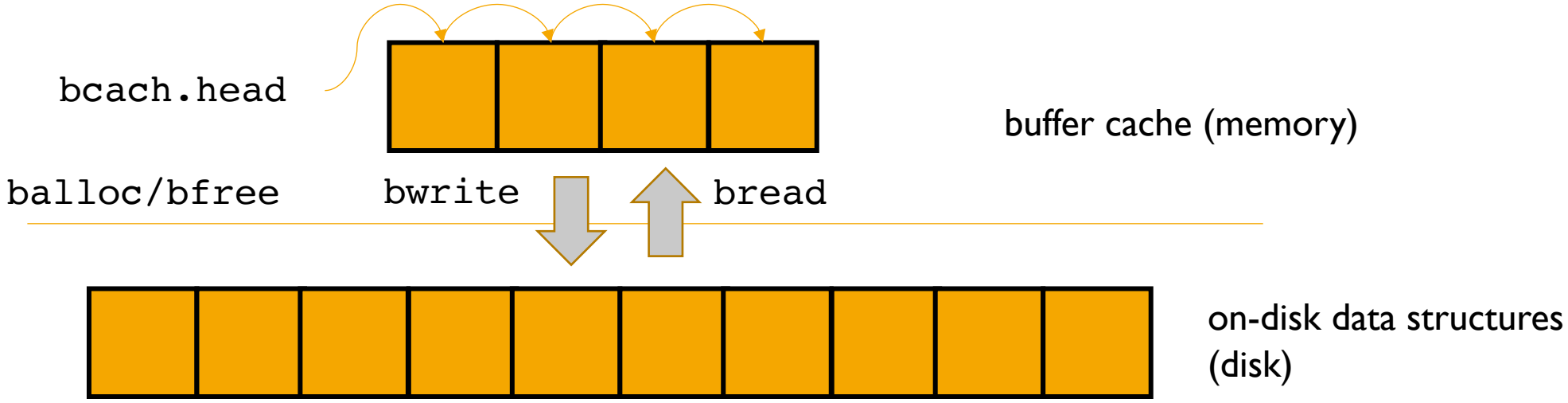



Figure 7.2: Structure of the xv6 file system.

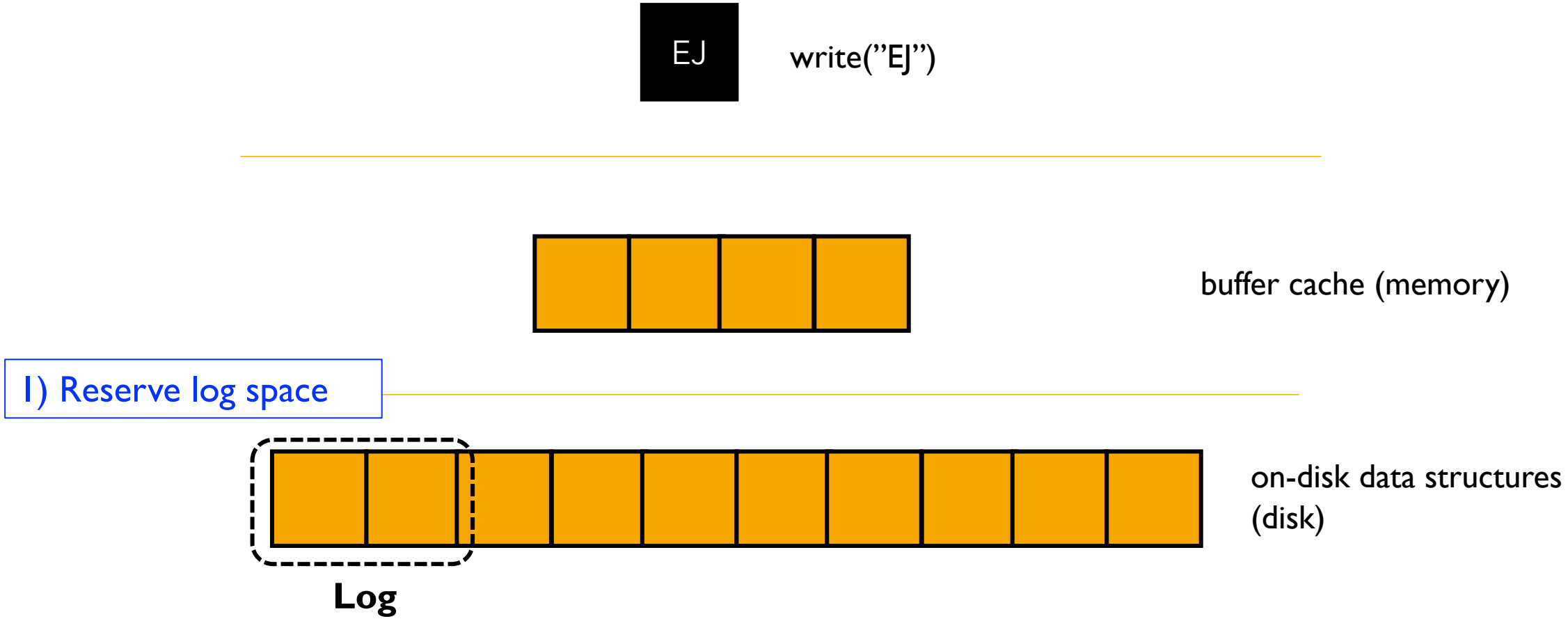
# Buffer Cache Layer

- `bio.c`
- Two jobs
  - 1) Synchronize access to disk blocks
  - 2) Cache popular blocks



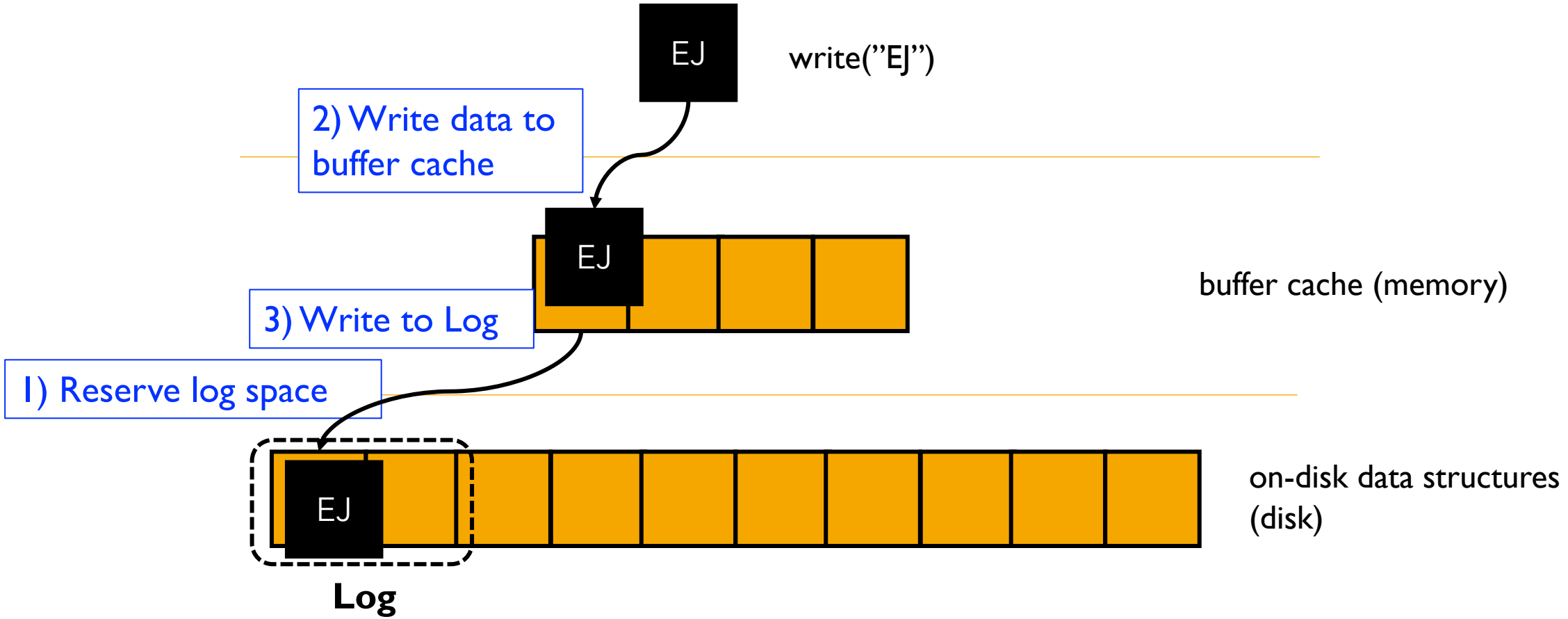
# Logging Layer

- `log.c`



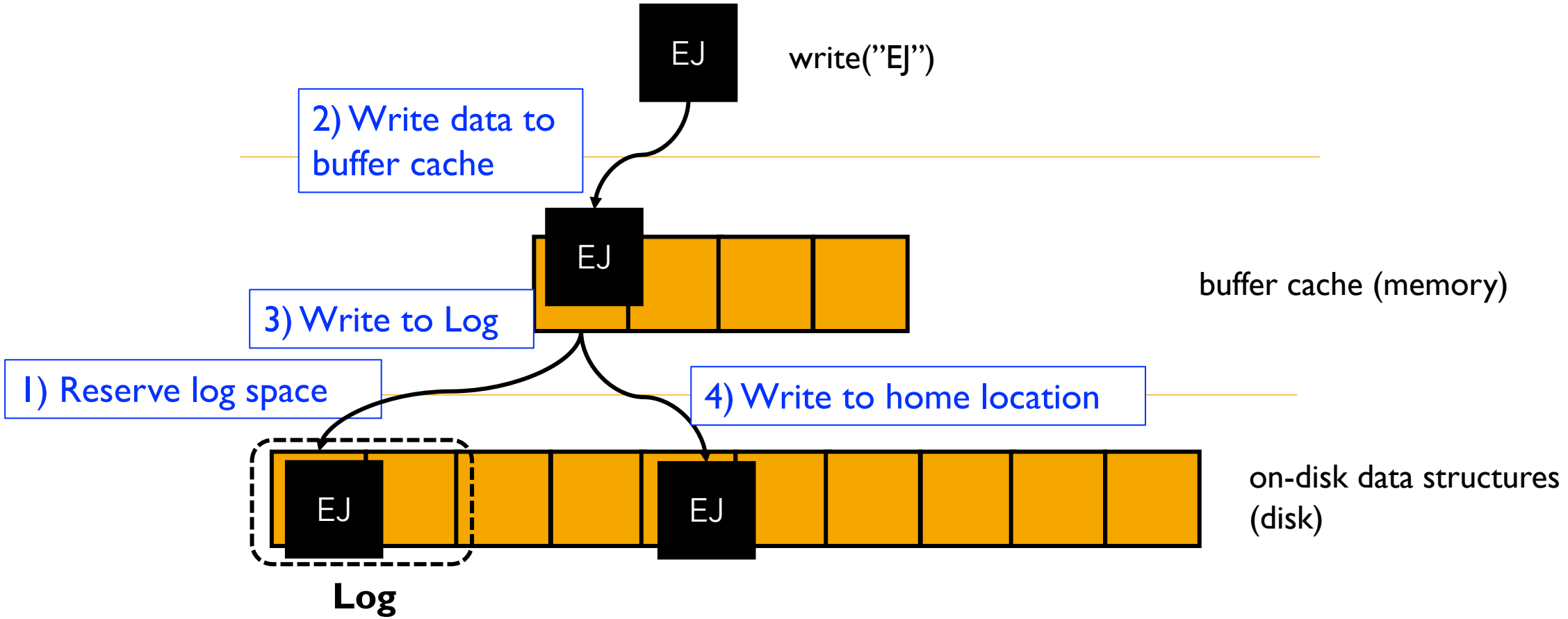
# Logging Layer

- `log.c`



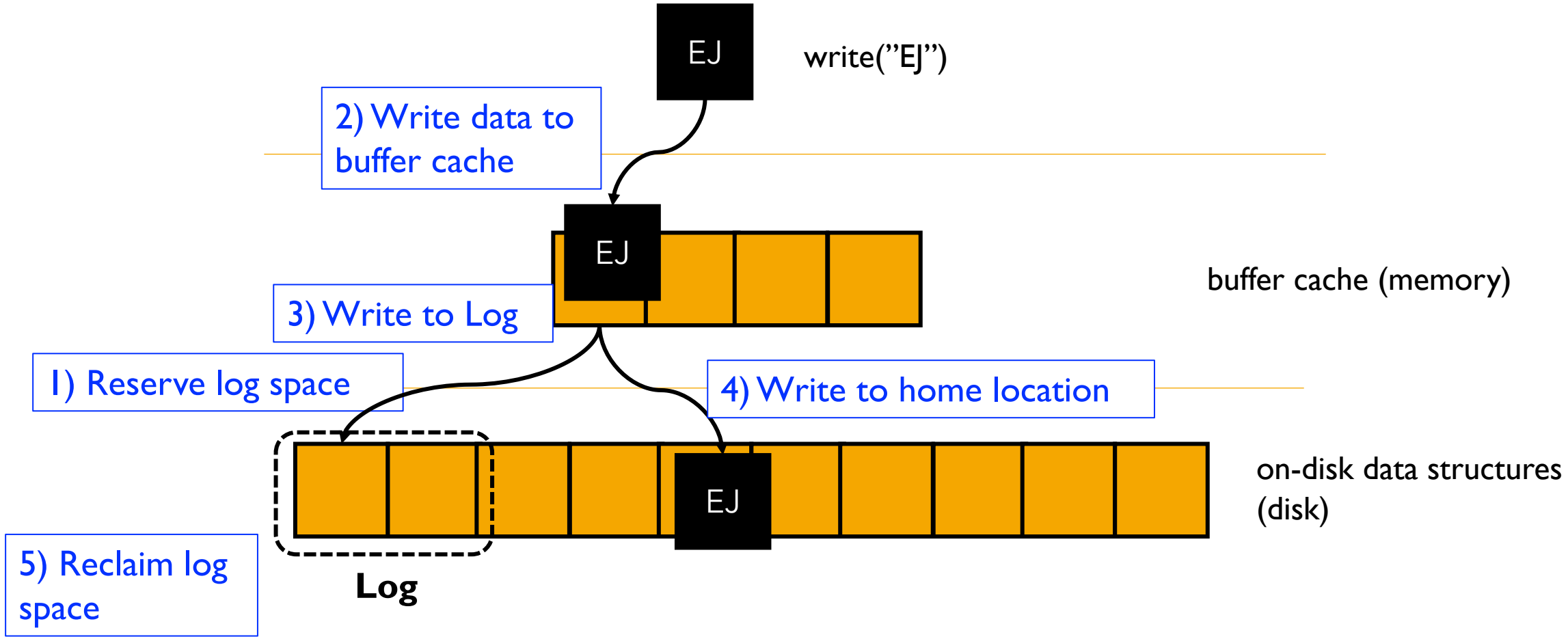
# Logging Layer

- `log.c`



# Logging Layer

- `log.c`

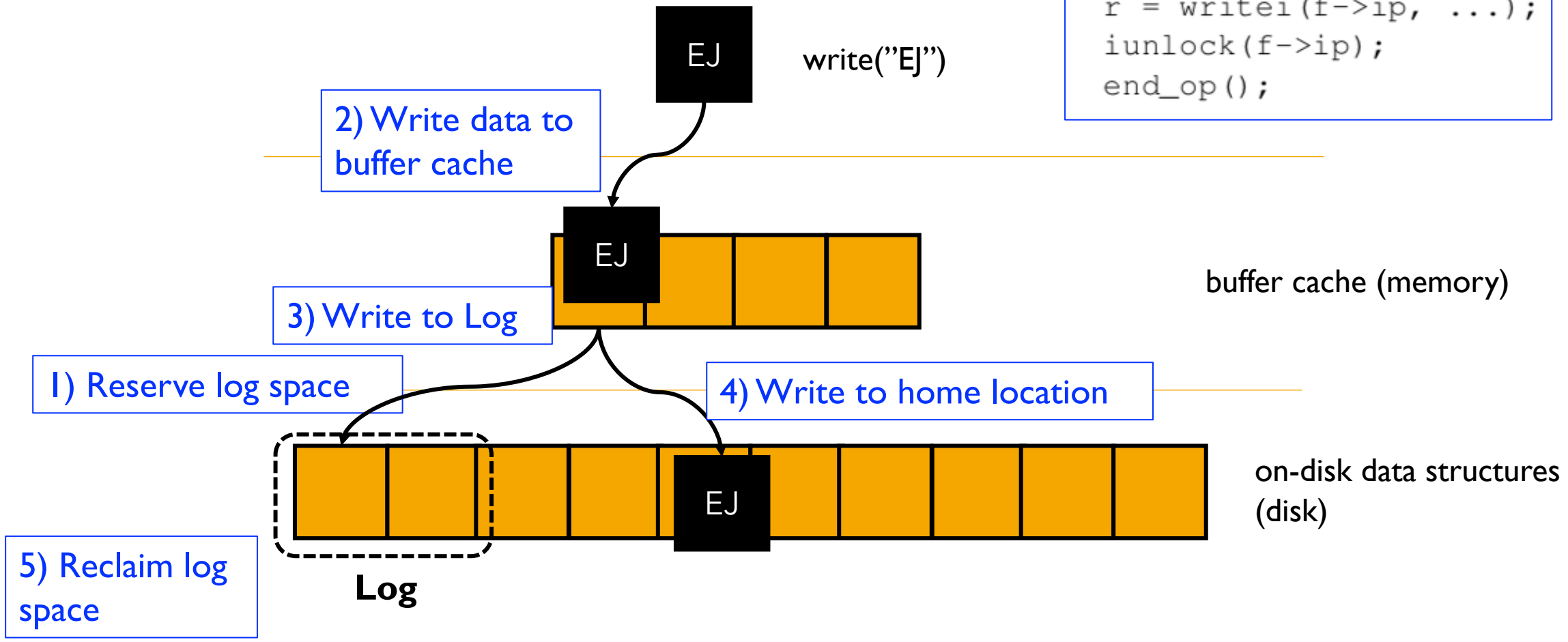


# Logging Layer

- `log.c`

`filewrite (file.c)`

```
begin_op();
ilock(f->ip);
r = writei(f->ip, ...);
iunlock(f->ip);
end_op();
```



# Inode Layer

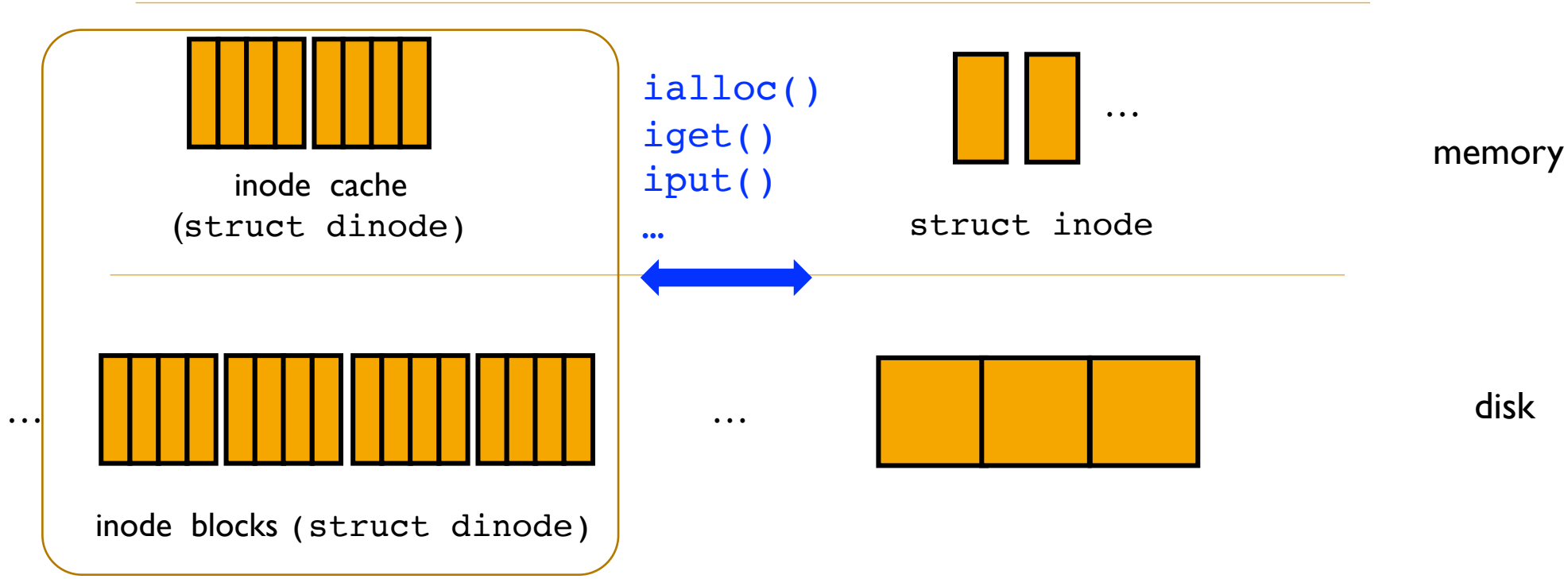
- `inode` has one of two related meanings
  - on-disk data structure
  - in-memory inode

- On-disk inode
  - `struct dinode` (fs.h)
  - Packed into a contiguous area of disk called inode blocks

- In-memory inode
  - struct inode (file.h)
  - in-memory copy of a `struct dinode`

# Inode Layer

- `fs.c`



# Inode Layer

- Inode contains pointers to data blocks

`fs.h`

```
24 #define NDIRECT 12
25 #define NINDIRECT (BSIZE / sizeof(uint))
26 #define MAXFILE (NDIRECT + NINDIRECT)
27
28 // On-disk inode structure
29 struct dinode {
30   short type;           // File type
31   short major;          // Major device number (T_DEV only)
32   short minor;          // Minor device number (T_DEV only)
33   short nlink;          // Number of links to inode in file system
34   uint size;            // Size of file (bytes)
35   uint addrs[NDIRECT+1];   // Data block addresses
36 };
37
38 // Inodes per block.
39 #define IPB           (BSIZE / sizeof(struct dinode))
40
41 // Block containing inode i
42 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
43
44 // Bitmap bits per block
45 #define BPB           (BSIZE*8)
46
47 // Block of free map containing bit for block b
48 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
```

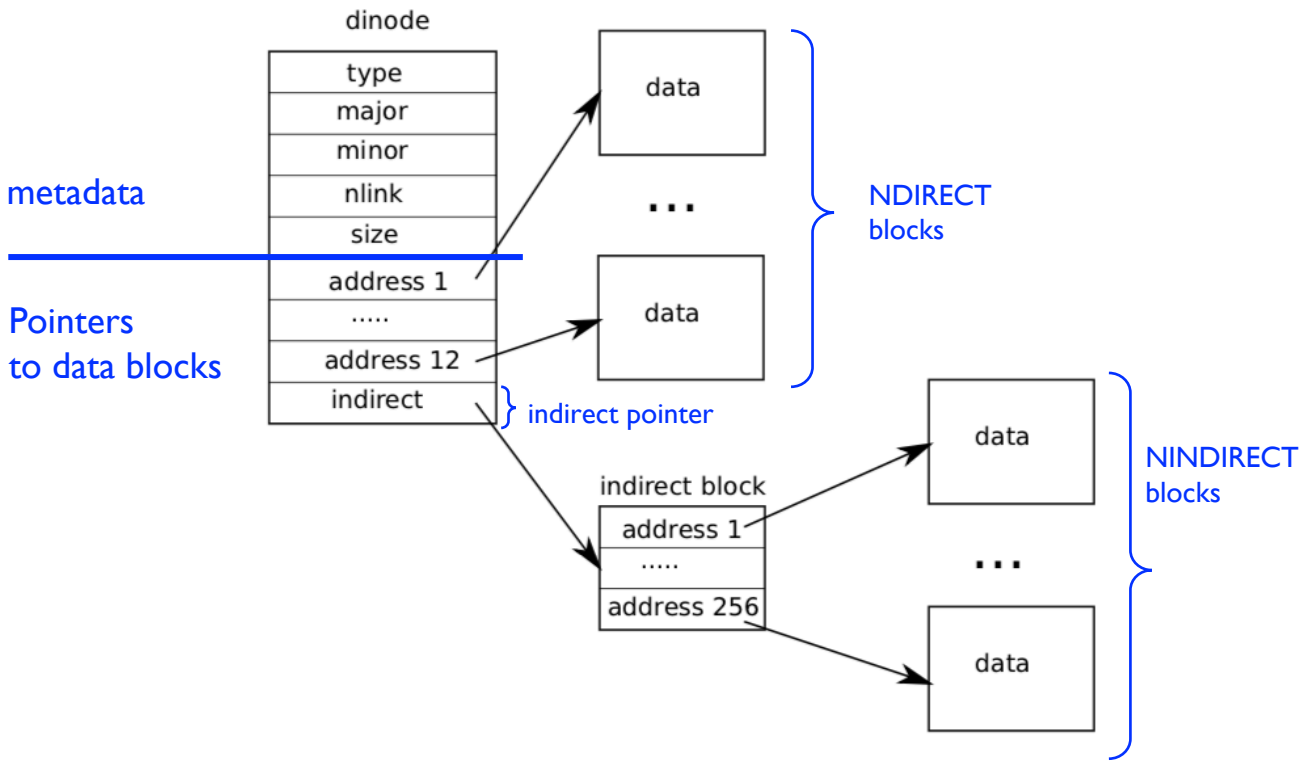



Figure 7.3: The representation of a file on disk.

# Inode Layer

- Inode contains pointers to data blocks

`fs.h`

```
24 #define NDIRECT 12
25 #define NINDIRECT (BSIZE / sizeof(uint))
26 #define MAXFILE (NDIRECT + NINDIRECT)
27
28 // On-disk inode structure
29 struct dinode {
30   short type;           // File type
31   short major;          // Major device number (T_DEV only)
32   short minor;          // Minor device number (T_DEV only)
33   short nlink;          // Number of links to inode in file system
34   uint size;            // Size of file (bytes)
35   uint addrs[NDIRECT+1];   // Data block addresses
36 };
37
38 // Inodes per block.
39 #define IPB           (BSIZE / sizeof(struct dinode))
40
41 // Block containing inode i
42 #define IBLOCK(i, sb)     ((i) / IPB + sb.inodestart)
43
44 // Bitmap bits per block
45 #define BPB           (BSIZE*8)
46
47 // Block of free map containing bit for block b
48 #define BBLOCK(b, sb) (b/BPB + sb.bmapstart)
```

```
5 #define ROOTINO 1  // root i-number
6 #define BSIZE 512  // block size
```

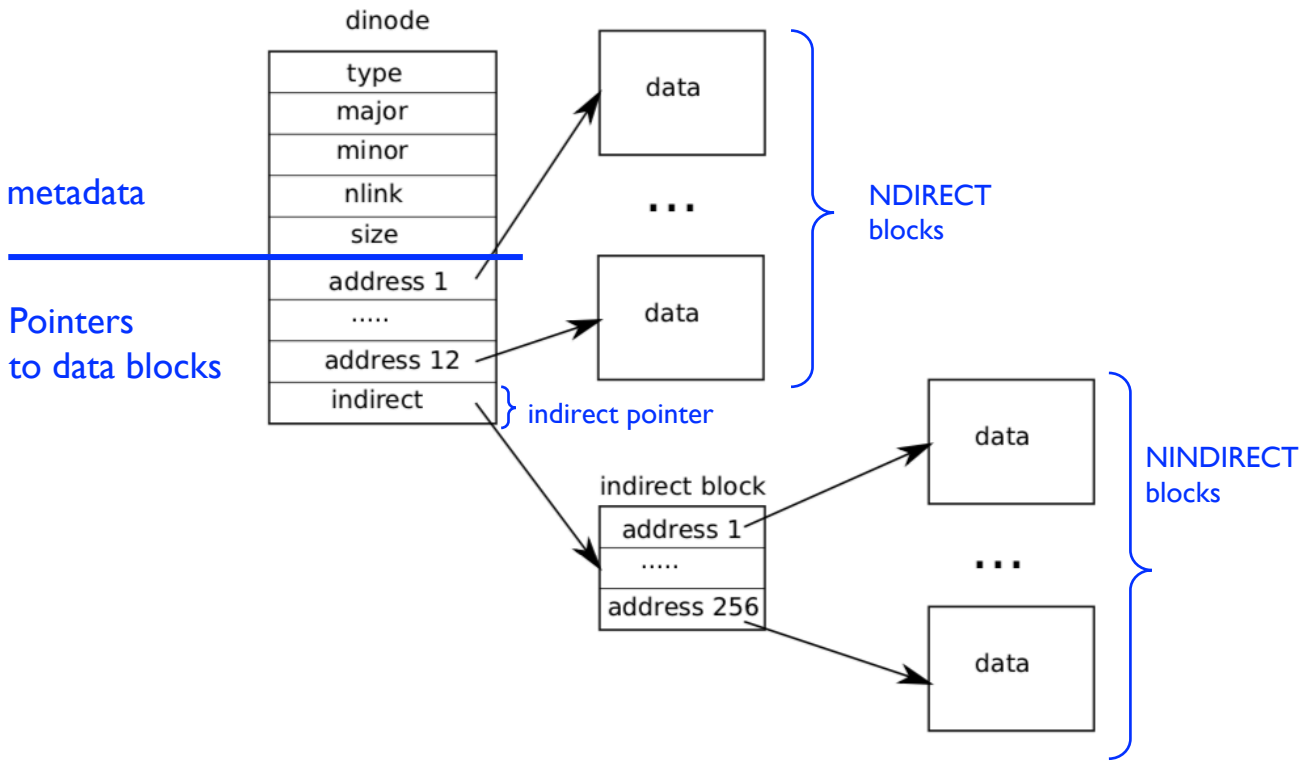



Figure 7.3: The representation of a file on disk.

# Inode Layer

- File operation through inode (fs.c)
  - `readi` : read data from file
  - `writei` : write data to file
  - `bmap` : return the corres logical block number of data block at offset

```
449 //PAGEBREAK!
450 // Read data from inode.
451 // Caller must hold ip->lock.
452 int
453 readi(struct inode *ip, char *dst, uint off, uint n)
454 {
455   uint tot, m;
456   struct buf *bp;
457
458   if(ip->type == T_DEV){
459     if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
460       return -1;
461     return devsw[ip->major].read(ip, dst, n);
462   }
463
464   if(off > ip->size || off + n < off)
465     return -1;
466   if(off + n > ip->size)
467     n = ip->size - off;
468
469   for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
470     bp = bread(ip->dev, bmap(ip, off/BSIZE));
471     m = min(n - tot, BSIZE - off%BSIZE);
472     memmove(dst, bp->data + off%BSIZE, m);
473     brelse(bp);
474   }
475   return n;
476 }
```

# Inode Layer

- File operation through inode (fs.c)
  - `readi` : read data from file
  - `writei` : write data to file
  - `bmap` : return the corres logical block number of data block at offset

```
362 //PAGEBREAK!
363 // Inode content
364 //
365 // The content (data) associated with each inode is stored
366 // in blocks on the disk. The first NDIRECT block numbers
367 // are listed in ip->addrs[].  The next NINDIRECT blocks are
368 // listed in block ip->addrs[NDIRECT].
369
370 // Return the disk block address of the nth block in inode ip.
371 // If there is no such block, bmap allocates one.
372 static uint
373 bmap(struct inode *ip, uint bn)
374 {
375   uint addr, *a;
376   struct buf *bp;
377
378   if(bn < NDIRECT){
379     if((addr = ip->addrs[bn]) == 0)
380       ip->addrs[bn] = addr = balloc(ip->dev);
381     return addr;
382   }
383   bn -= NDIRECT;
384
385   if(bn < NINDIRECT){
386     // Load indirect block, allocating if necessary.
387     if((addr = ip->addrs[NDIRECT]) == 0)
388       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
389     bp = bread(ip->dev, addr);
390     a = (uint*)bp->data;
391     if((addr = a[bn]) == 0){
392       a[bn] = addr = balloc(ip->dev);
393       log_write(bp);
394     }
395     brelse(bp);
396     return addr;
397   }
398
399   panic("bmap: out of range");
400 }
```

logical block number (in disk)

# Directory Layer

- Directory is implemented like a file
- Inode has type `T_DIR`
- Data contains a sequence of `dirent` structures
- Directory operations (fs.c)
  - `dirlookup` : search a directory for an entry with the given name
  - `dirlink` : write a new directory entry with the given name and inode number to the directory

`fs.h`

```
51 #define DIRSIZ 14
52
53 struct dirent {
54   ushort inum;
55   char name[DIRSIZ];
56 };
57
```

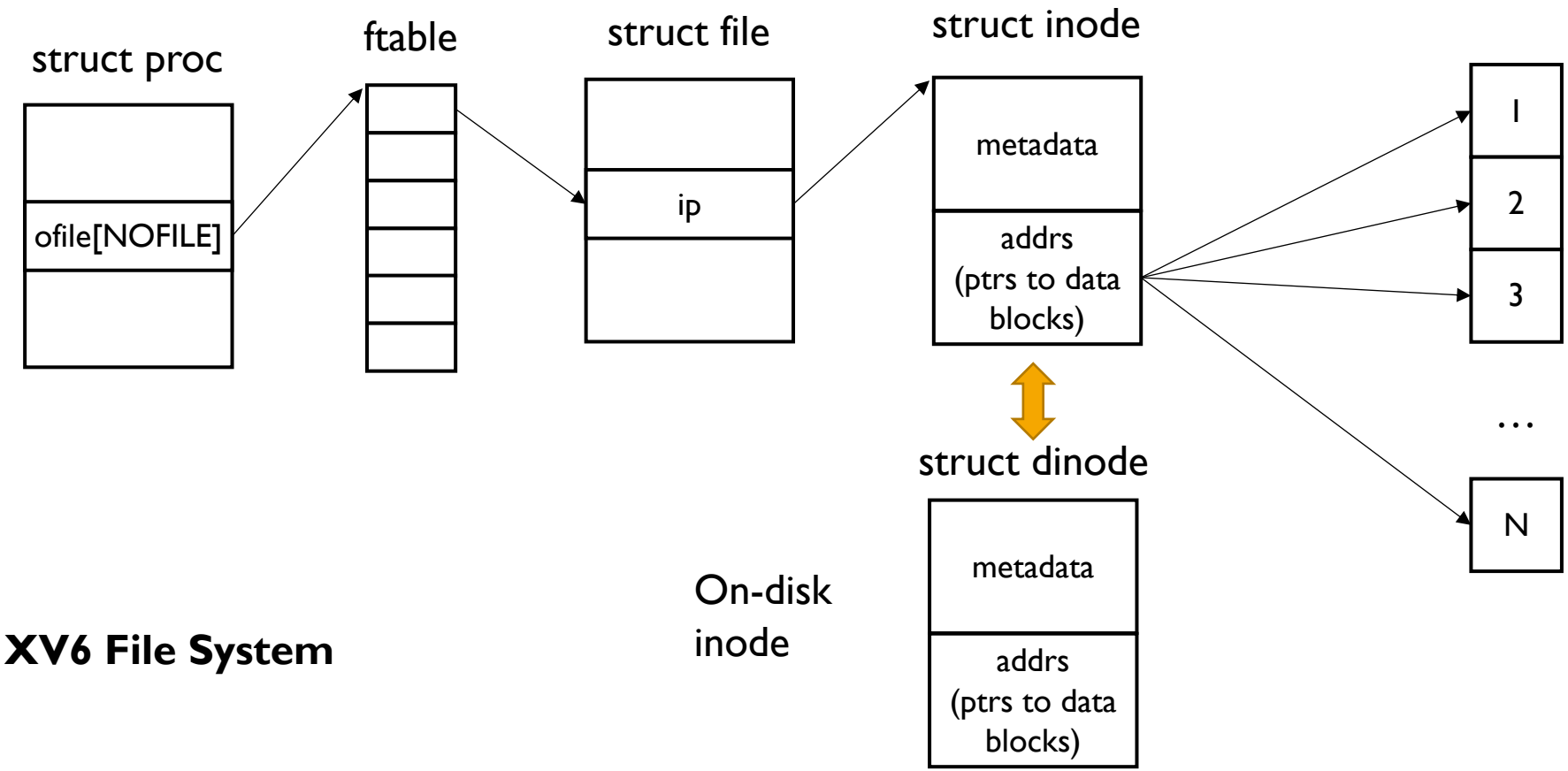
# Pathname Layer

- Involve a succession of calls to dirlookup
- Pathname operations
  - `namei` : evaluate path and returns the corresponding inode

```
620 // Look up and return the inode for a path name.
621 // If parent != 0, return the inode for the parent and copy the final
622 // path element into name, which must have room for DIRSIZ bytes.
623 // Must be called inside a transaction since it calls iput().
624 static struct inode*
625 namex(char *path, int nameiparent, char *name)
626 {
627   struct inode *ip, *next;
628
629   if(*path == '/')
630     ip = iget(ROOTDEV, ROOTINO);
631   else
632     ip = idup(myproc()->cwd);
633
634   while((path = skipelem(path, name)) != 0){
635     ilock(ip);
636     if(ip->type != T_DIR){
637       iunlockput(ip);
638       return 0;
639     }
640     if(nameiparent && *path == '\0'){
641       // Stop one level early.
642       iunlock(ip);
643       return ip;
644     }
645     if((next = dirlookup(ip, name, 0)) == 0){
646       iunlockput(ip);
647       return 0;
648     }
649     iunlockput(ip);
650     ip = next;
651   }
652   if(nameiparent){
653     iput(ip);
654     return 0;
655   }
656   return ip;
657 }
658
659 struct inode*
660 namei(char *path)
661 {
662   char name[DIRSIZ];
663   return namex(path, 0, name);
664 }
665
666 struct inode*
667 nameiparent(char *path, char *name)
668 {
669   return namex(path, 1, name);
670 }
```

# File descriptor Layer

- Most resources are represented as files, including devices such as console, pipes, real files




**XV6 File System**

# File descriptor Layer

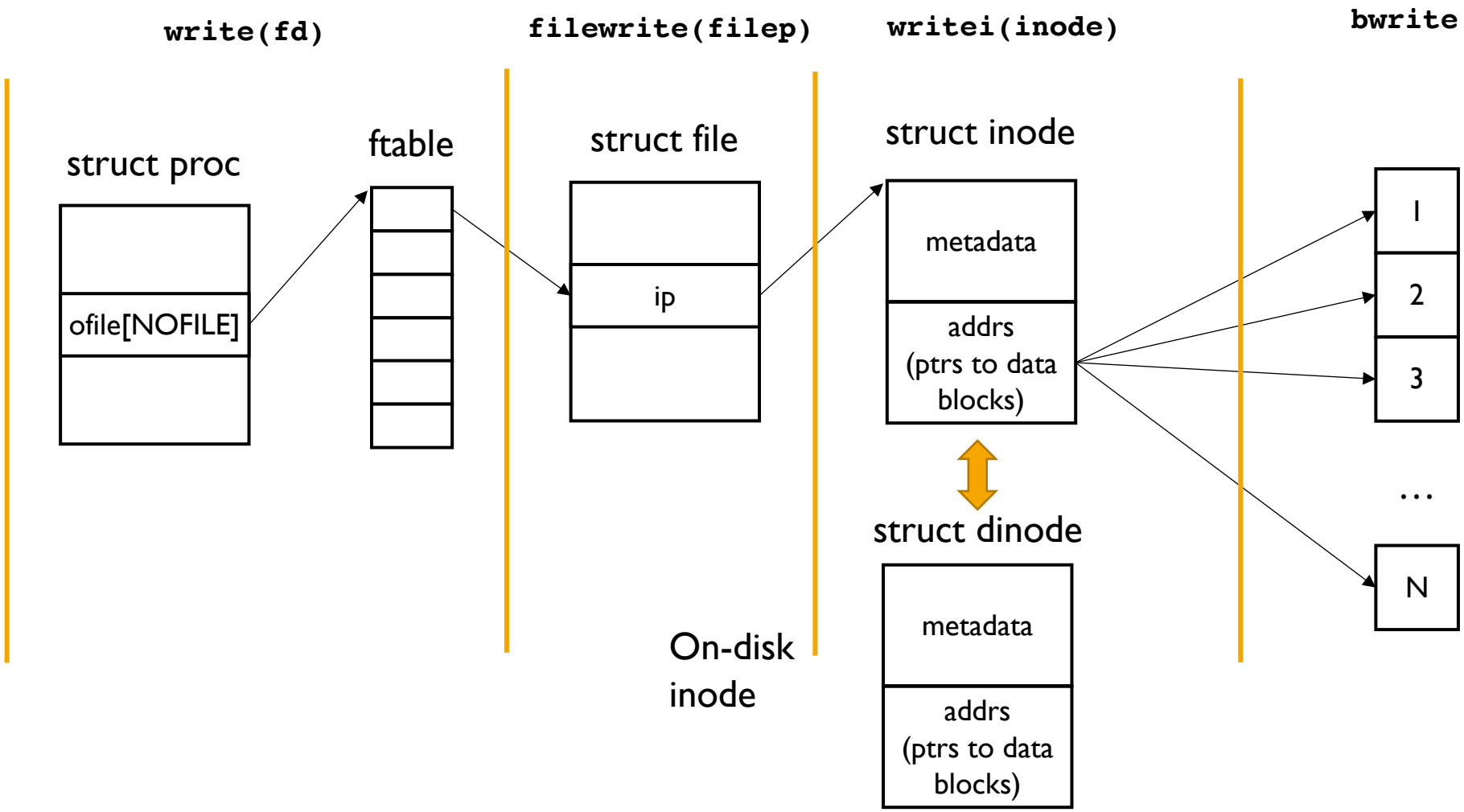
- File operations (file.c)
  - `struct file`
  - `filealloc`
  - `filedup`
  - `fileclose`
  - `fileread`
  - `filewrite`

# System call Layer

- System call for files (sysfile.c)
  - `sys_link / sys_unlink`
  - `sys_open`
  - `sys_dup`
  - `sys_read`
  - `sys_write`
  - `sys_mkdir`
  - `sys_chdir`
  - `sys_close`

# File Operations

- XV6 File System



# Project 6. File extension

- Maximum file size is140 sectors (70 KB) in XV6
- Support a large file size by using the “doubly-indirect block”
- Modified file system will allow 16523 sectors for each file
- Codes to be modified
  - `bmap()` in fs.c
  - Associated settings (in fs.h file.h, etc. )
  - file system size (in param.h)

# Project 6. File extension

- `fs.h`

```
24 // BIG_FILE
25 #define NDIRECT 11
26 #define NINDIRECT (BSIZE / sizeof(uint))
27 #define NDINDIRECT NINDIRECT*NINDIRECT
28 #define MAXFILE (NDIRECT + NINDIRECT + NDINDIRECT)
29 //#define NDIRECT 12
30 //#define NINDIRECT (BSIZE / sizeof(uint))
31 //#define MAXFILE (NDIRECT + NINDIRECT)
32
33 // On-disk inode structure
34 struct dinode {
35   short type;           // File type
36   short major;          // Major device number (T_DEV only)
37   short minor;          // Minor device number (T_DEV only)
38   short nlink;          // Number of links to inode in file system
39   uint size;            // Size of file (bytes)
40   uint addrs[NDIRECT+2];   // Data block addresses // BIG_FILE
41 //  uint addrs[NDIRECT+1];   // Data block addresses
42 };
```

# Project 6. File extension

- `file.h`

```
12 // in-memory copy of an inode
13 struct inode {
14   uint dev;           // Device number
15   uint inum;          // Inode number
16   int ref;            // Reference count
17   struct sleeplock lock; // protects everything below here
18   int valid;          // inode has been read from disk?
19
20   short type;         // copy of disk inode
21   short major;
22   short minor;
23   short nlink;
24   uint size;
25   uint addrs[NDIRECT+2]; // BIG_FILEE
26 //  uint addrs[NDIRECT+1];
27 };
```

# Project 6. File extension

- `fs.c`

```
362 //PAGEBREAK!
363 // Inode content
364 //
365 // The content (data) associated with each inode is stored
366 // in blocks on the disk. The first NDIRECT block numbers
367 // are listed in ip->addrs[].  The next NINDIRECT blocks are
368 // listed in block ip->addrs[NDIRECT].
369
370 // Return the disk block address of the nth block in inode ip.
371 // If there is no such block, bmap allocates one.
372 static uint
373 bmap(struct inode *ip, uint bn)
374 {
375   uint addr, *a;
376   struct buf *bp;
377
378   if(bn < NDIRECT){
379     if((addr = ip->addrs[bn]) == 0)
380       ip->addrs[bn] = addr = balloc(ip->dev);
381     return addr;
382   }
383   bn -= NDIRECT;
384
385   if(bn < NINDIRECT){
386     // Load indirect block, allocating if necessary.
387     if((addr = ip->addrs[NDIRECT]) == 0)
388       ip->addrs[NDIRECT] = addr = balloc(ip->dev);
389     bp = bread(ip->dev, addr);
390     a = (uint*)bp->data;
391     if((addr = a[bn]) == 0){
392       a[bn] = addr = balloc(ip->dev);
393       log_write(bp);
394     }
395     brelse(bp);
396     return addr;
397   }
398
399   panic("bmap: out of range");
400 }
```

add codes here!

# Project 6. File extension

- `bigfiletest.c`

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int
main()
{
  char buf[512];
  int fd, i, sectors;

  fd = open("big.file", O_CREATE | O_WRONLY);
  if(fd < 0){
    printf(2, "big: cannot open big.file for writing\n");
    exit();
  }

  sectors = 0;
  while(1){
    *(int*)buf = sectors;
    int cc = write(fd, buf, sizeof(buf));
    if(cc <= 0)
      break;
    sectors++;
  if (sectors % 100 == 0)
    printf(2, ".");
  }

  printf(1, "\nwrote %d sectors\n", sectors);

  close(fd);
  fd = open("big.file", O_RDONLY);
  if(fd < 0){
    printf(2, "big: cannot re-open big.file for reading\n");
    exit();
  }
  for(i = 0; i < sectors; i++){
    int cc = read(fd, buf, sizeof(buf));
    if(cc <= 0){
      printf(2, "big: read error at sector %d\n", i);
      exit();
    }
    if(*(int*)buf != i){
      printf(2, "big: read the wrong data (%d) for sector %d\n",
             *(int*)buf, i);
      exit();
    }
  }

  printf(1, "done; ok\n");

  exit();
}

```

# Project 6. File extension

- `bigfiletest.c`

```
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ bigfiletest
.
wrote 140 sectors
done; ok
```

```
xv6...
cpu0: starting 0
sb: size 21113 nblocks 21049 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ bigfiletest
..........................................................................................
..........................
wrote 16523 sectors
done; ok
$
```

# Hand-in Procedures

- Download template
  - https://github.com/eunjicious/xv6-ssu.git
  - `tar xvzf xv6_ssu_fs.tar.gz`
- Modify codes
- Compress your code (ID: 20201234)
  - `$tar cvzf xv6_ssu_sched_20201234.tar.gz xv6_ssu_sched`
  - Please command `$make clean` before compressing
- Submit your `tar.gz` file through myclass.ssu.ac.kr
- NO DELAY is allowed !!
- PLEASE DO NOT COPY !!