File Systems

Vera Goebel
Thomas Plagemann

2014

Department of Informatics
University of Oslo

Todays Plan

Long-term Information Storage

1. Must store large amounts of data
2. Information stored must survive the termination of the process using it
3. Multiple processes must be able to access the information concurrently

3

Why Files?

- Physical reality
  - Block oriented
  - Physical sector numbers
  - No protection among users of the system
  - Data might be corrupted if machine crashes
- File system abstraction
  - Byte oriented
  - Named files
  - Users protected from each other
  - Robust to machine failures

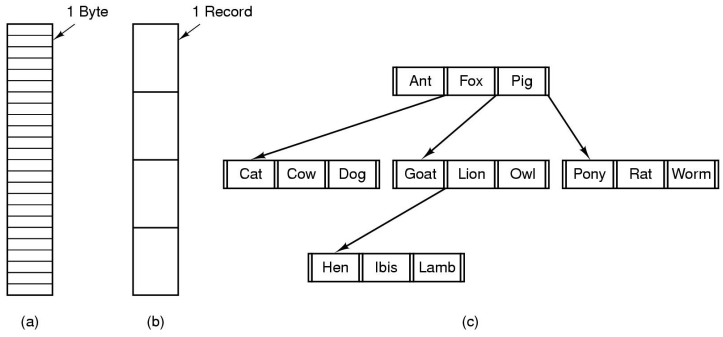
File Naming

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Typical file extensions.

5

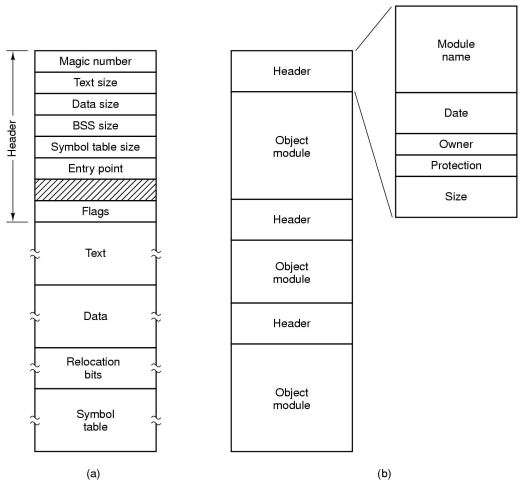
File Structure



- Three kinds of files
  - byte sequence
  - record sequence
  - tree

6

File Types



(a) An executable file (b) An archive

7

File Access

- Sequential access
  - read all bytes/records from the beginning
  - cannot jump around, could rewind or back up
  - convenient when medium was mag tape
- Random access
  - bytes/records read in any order
  - essential for data base systems
  - read can be …
    - move file marker (seek), then read or …
    - read and then move file marker

8

File Attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Possible file attributes

9

File Operations

1. Create
2. Delete
3. Open
4. Close
5. Read
6. Write
7. Append
8. Seek
9. Get attributes
10. Set Attributes
11. Rename

10

An Example Program Using File System Calls (1/2)

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* syntax error if argc is not 3 */
```

11

An Example Program Using File System Calls (2/2)

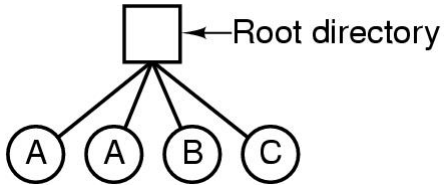
```
    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);    /* open the source file */
    if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
    if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;           /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* no error on last read */
        exit(0);
    else
        exit(5);                        /* error on last read */
}
```

12

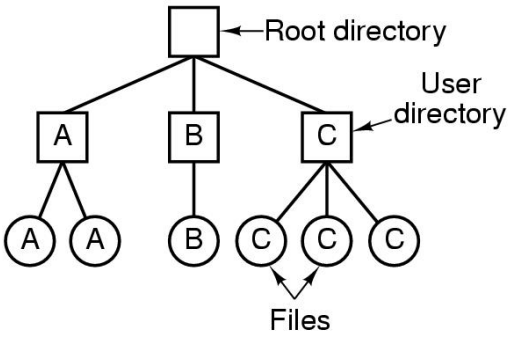
Directories
Single-Level Directory Systems



- A single level directory system
  - contains 4 files
  - owned by 3 different people, A, B, and C

13

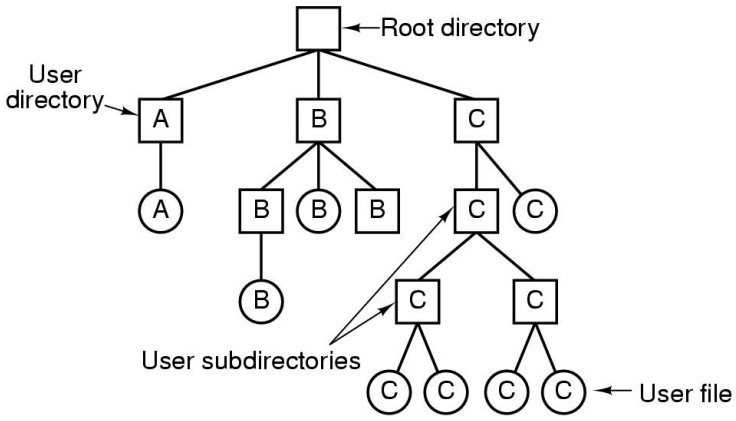
Two-level Directory Systems



Letters indicate *owners* of the directories and files

14

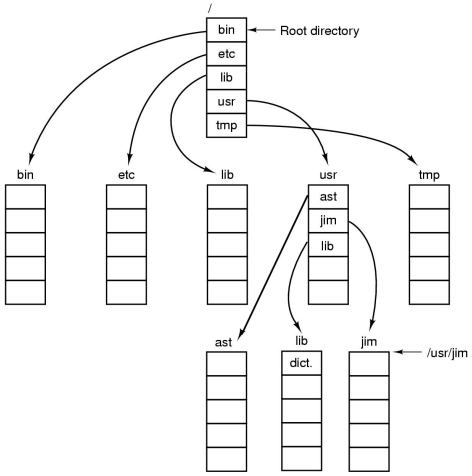
Hierarchical Directory Systems



A hierarchical directory system

15

Path Names



A UNIX directory tree

16

Directory Operations

1. Create
2. Delete
3. Opendir
4. Closedir
5. Readdir
6. Rename
7. Link
8. Unlink

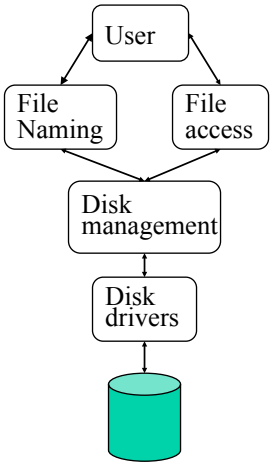
17

Now we are opening the box

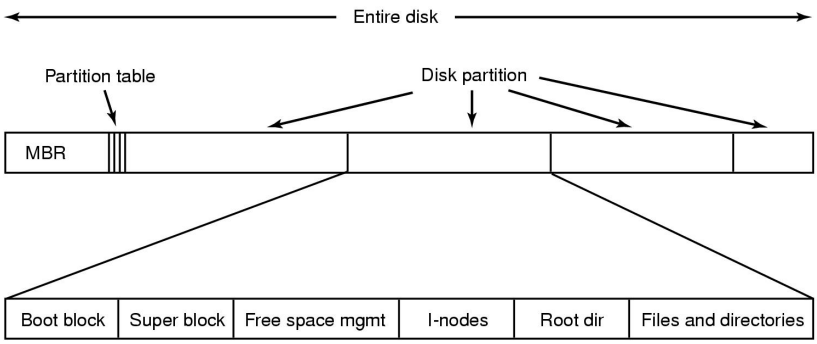
18

File System Components

- Disk management
  - Arrange collection of disk blocks into files
- Naming
  - User gives file name, not track or sector number, to locate data
- Security
  - Keep information secure
- Reliability/durability
  - When system crashes, lose stuff in memory, but want files to be durable



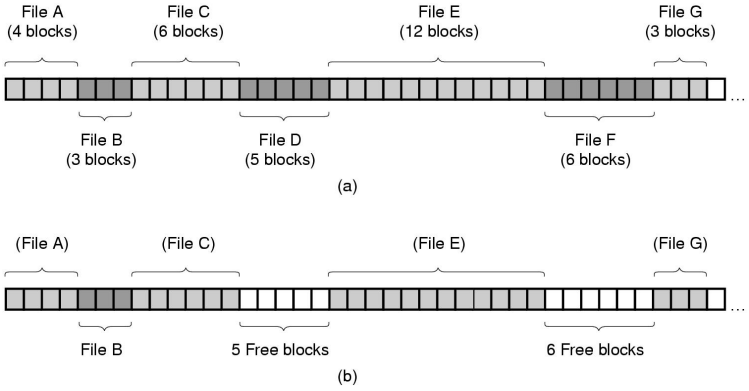
File System Implementation



A possible file system layout

20

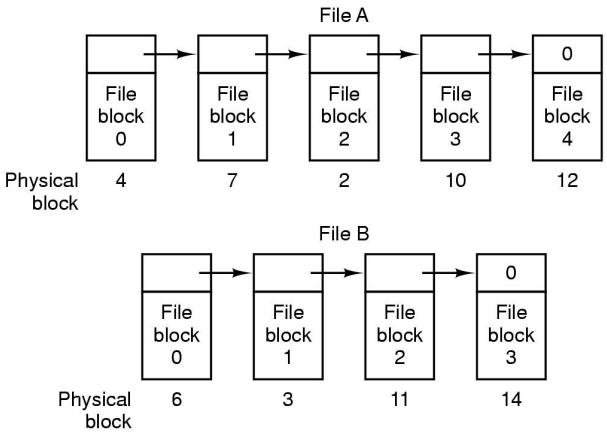
Implementing Files (1)



(a) Contiguous allocation of disk space for 7 files
(b) State of the disk after files *D* and *E* have been removed

21

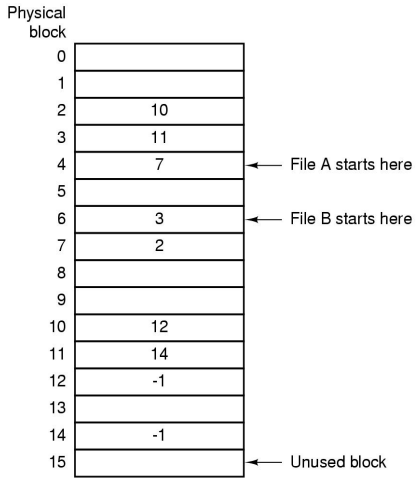
Implementing Files (2)



Storing a file as a linked list of disk blocks

22

Implementing Files (3)

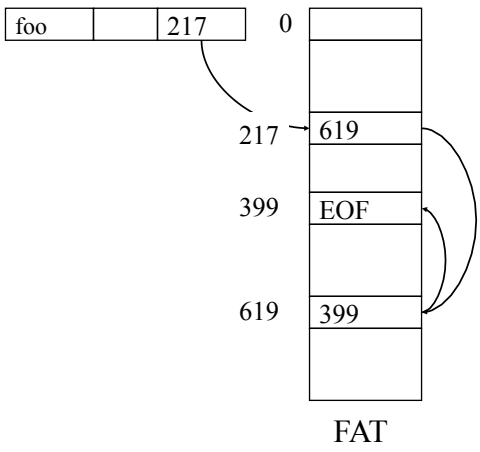


Linked list allocation using a file allocation table in RAM

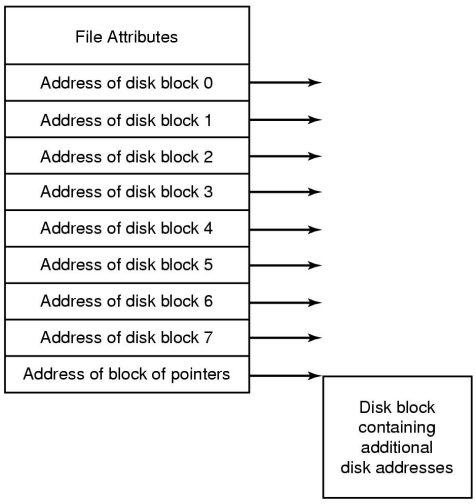
23

File Allocation Table (FAT)

- Approach
  - A section of disk for each partition is reserved
  - One entry for each block
  - A file is a linked list of blocks
  - A directory entry points to the 1st block of the file
- Pros
  - Simple
- Cons
  - Always go to FAT
  - Wasting space



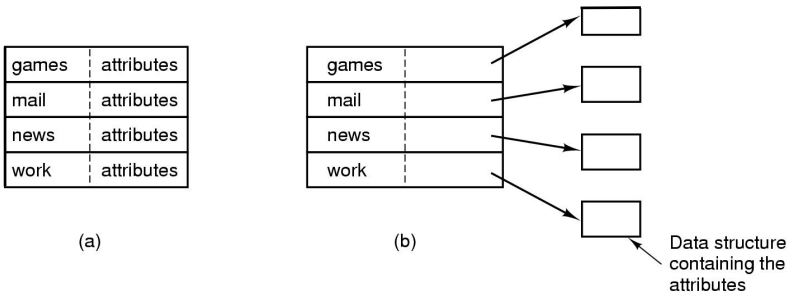
Implementing Files (4)



An example i-node

25

Implementing Directories (1)



(a) A simple directory

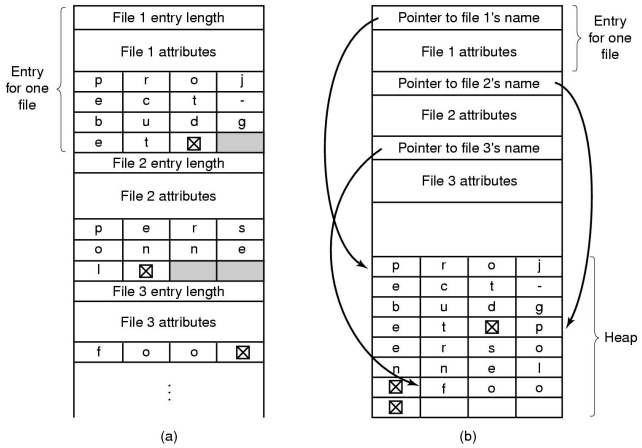
fixed size entries

disk addresses and attributes in directory entry

(b) Directory in which each entry just refers to an i-node

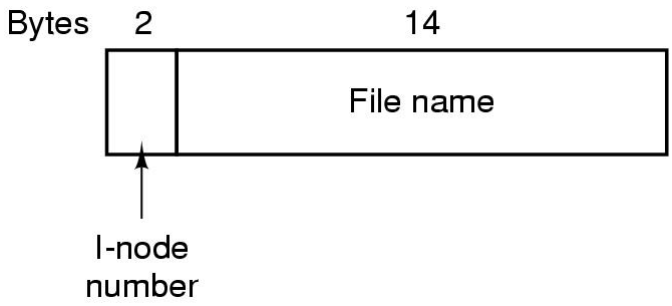
26

Implementing Directories (2)



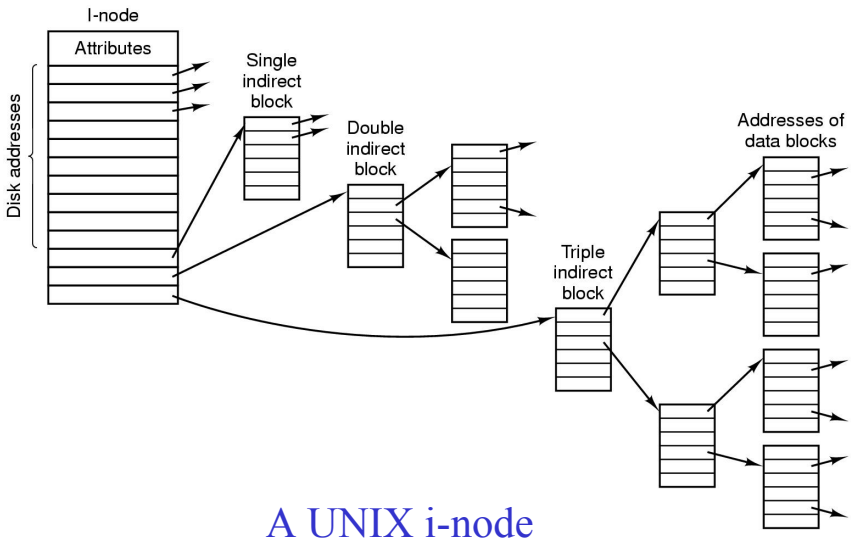
- Two ways of handling long file names in directory
  - (a) In-line
  - (b) In a heap

The UNIX V7 File System (1)



A UNIX V7 directory entry

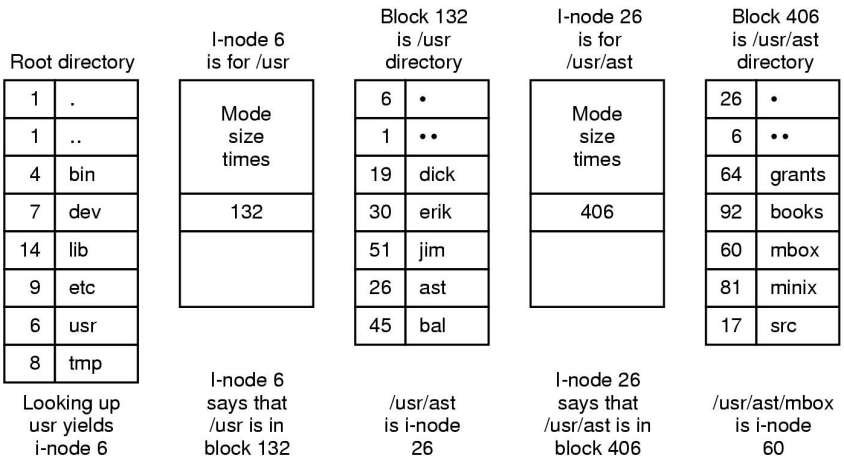
The UNIX V7 File System (2)



A UNIX i-node

29

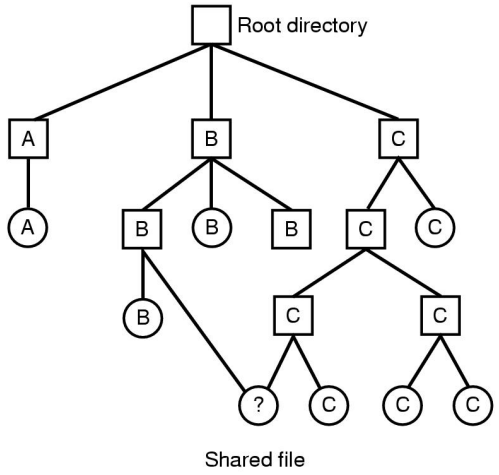
The UNIX V7 File System (3)



The steps in looking up */usr/ast/mbox*

30

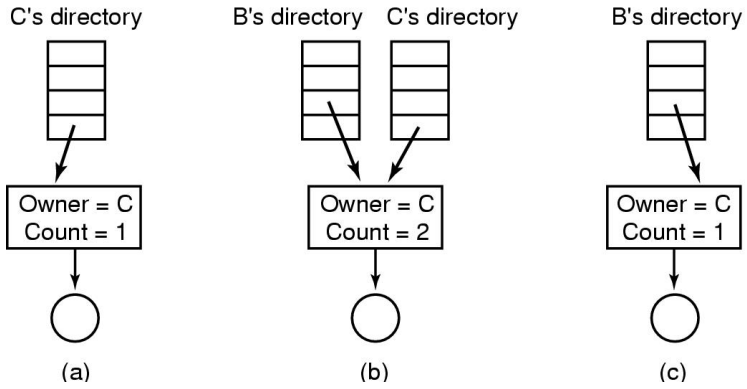
Shared Files (1)



File system containing a shared file

31

Shared Files (2)



(a) Situation prior to linking

(b) After the link is created

(c)After the original owner removes the file

32

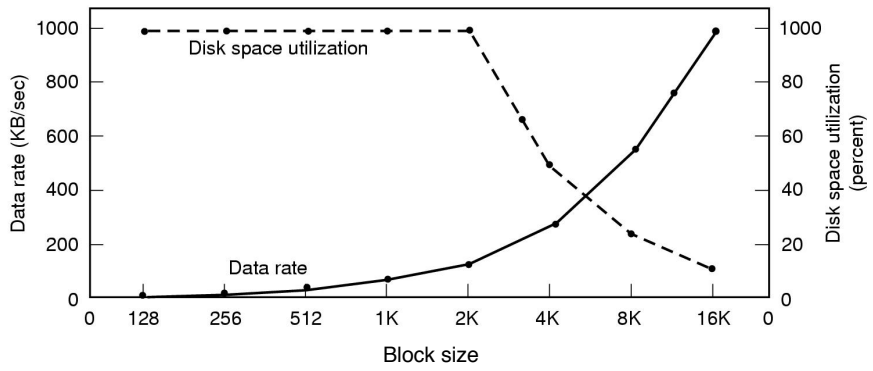
Block Size

Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1024	48.05	30.91	47.82
2048	60.87	46.09	59.44
4096	73.51	59.13	70.64
8192	84.97	69.96	79.69

Length	VU 1984	VU 2005	Web
16,384	92.53	78.92	86.79
32,768	97.21	85.87	91.65
65,536	99.18	90.84	94.80
131,072	99.84	93.73	96.93
262,144	99.96	96.12	98.48
524,288	100.00	97.73	98.99
1,048,576	100.00	98.87	99.62
2,097,162	100.00	99.44	99.80
4,194,304	100.00	99.71	99.87
8,388,608	100.00	99.86	99.94
16,777,216	100.00	99.94	99.97
33,554,432	100.00	99.97	99.99
67,108,864	100.00	99.99	99.99
134,217,728	100.00	99.99	100.00

33

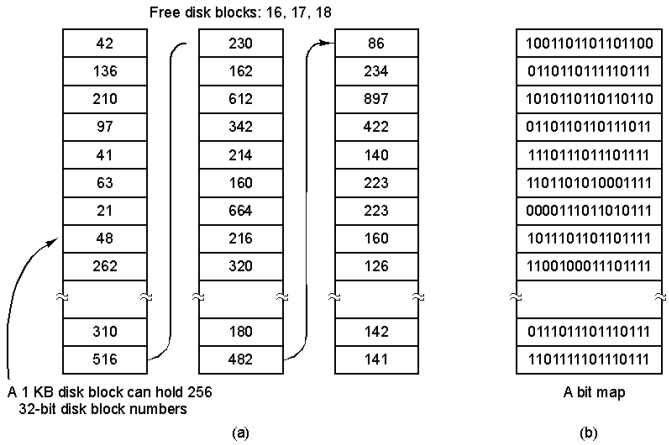
Disk Space Management (1)



- Dark line (left hand scale) gives data rate of a disk
- Dotted line (right hand scale) gives disk space efficiency
- All files 2KB

34

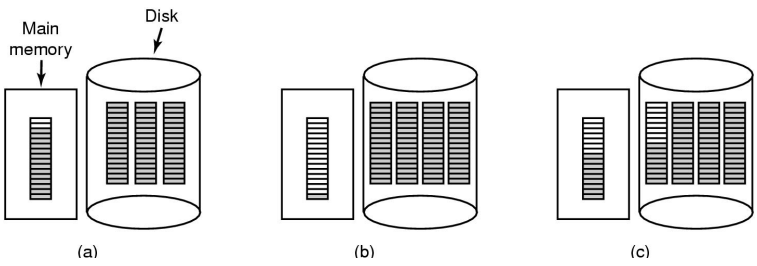
Disk Space Management (2)



(a) Storing the free list on a linked list

(b) A bit map

Disk Space Management (3)

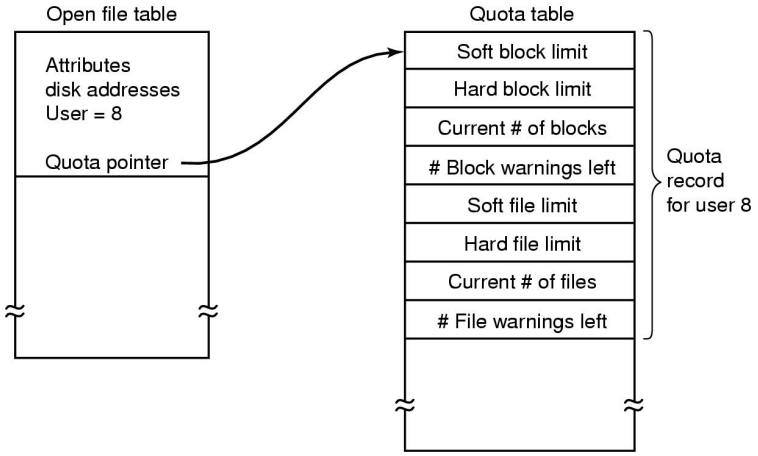


(a) Almost-full block of pointers to free disk blocks in RAM
- three blocks of pointers on disk

(b) Result of freeing a 3-block file

(c) Alternative strategy for handling 3 free blocks
- shaded entries are pointers to free disk blocks

Disk Space Management (4)



Quotas for keeping track of each user's disk use

37

How many FSs do we have on one PC?

38

Virtual File System

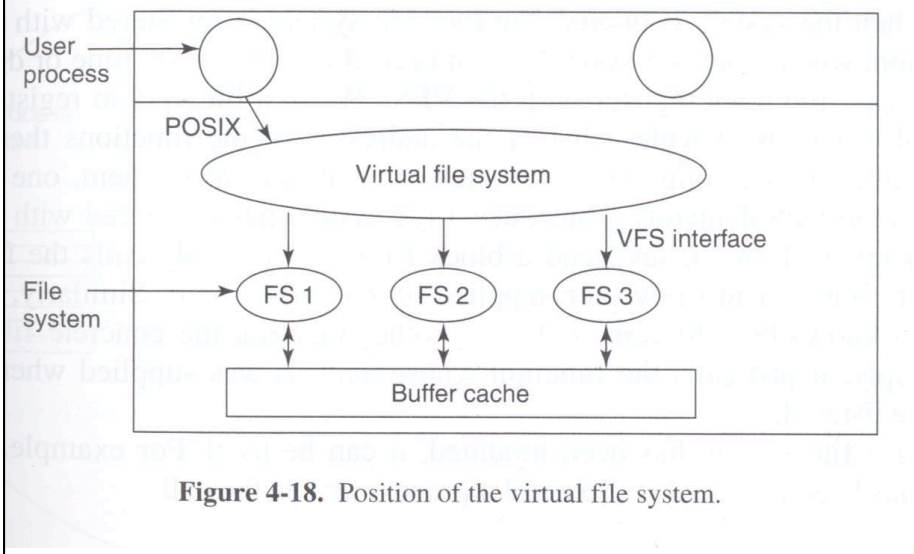


Figure 4-18. Position of the virtual file system.

39

Creating a VFS

- Boot time of OS: register root FS with VFS
- Mounting of others FSs (boot time or later): register with VFS
- Registering: provide function pointers to the FS specific calls

40

Using a FS via VFS

- `Open("/usr/include/unistd.h", O_RDONLY)`

41

VFS Structure

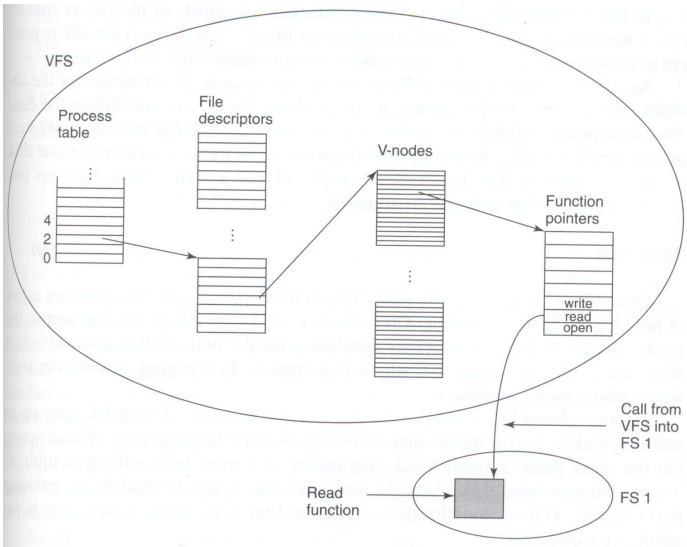


Figure 4-19. A simplified view of the data structures and code used by the VFS and concrete file system to do a read.

42

Network File System (NFS)

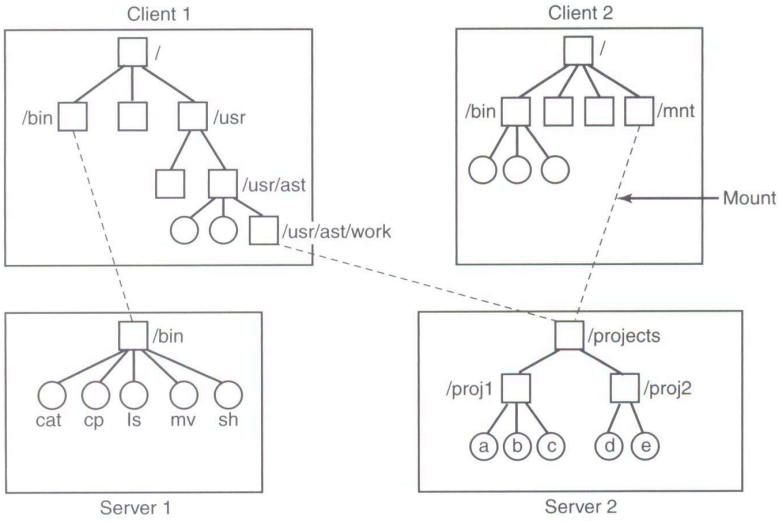
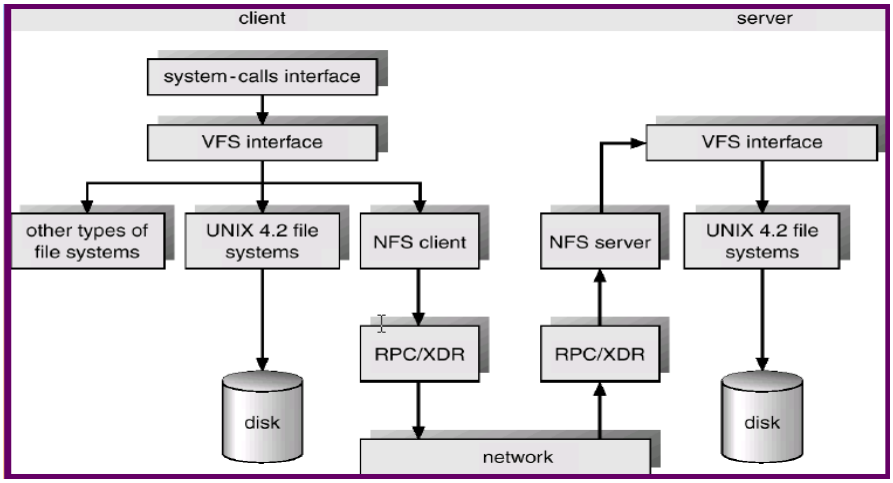


Figure 10-35. Examples of remote mounted file systems. Directories are shown as squares and files are shown as circles.

43

NFS Architecture

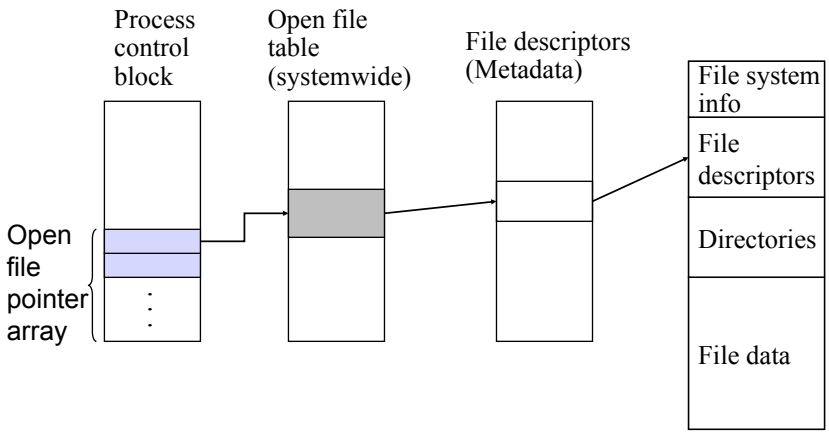


44

A Look into Linux-like "Mechanics"

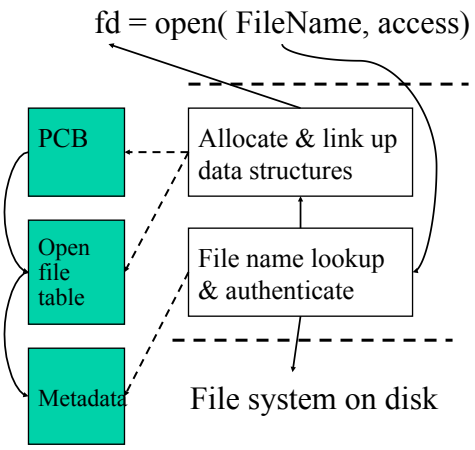
45

File System Data Structures

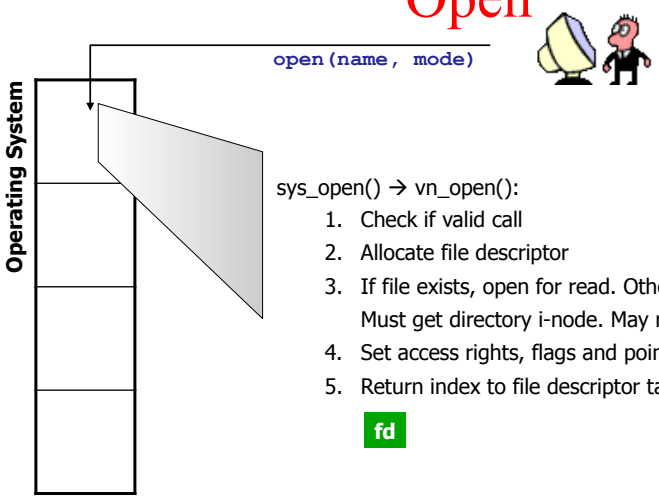


Open File

- File name lookup and authenticate
- Copy the file descriptors into the in memory data structure, if it is not in yet
- Create an entry in the open file table (system wide) if there isn't one
- Create an entry in PCB
- Link up the data structures
- Return a pointer to user



Open



sys_open() → vn_open():

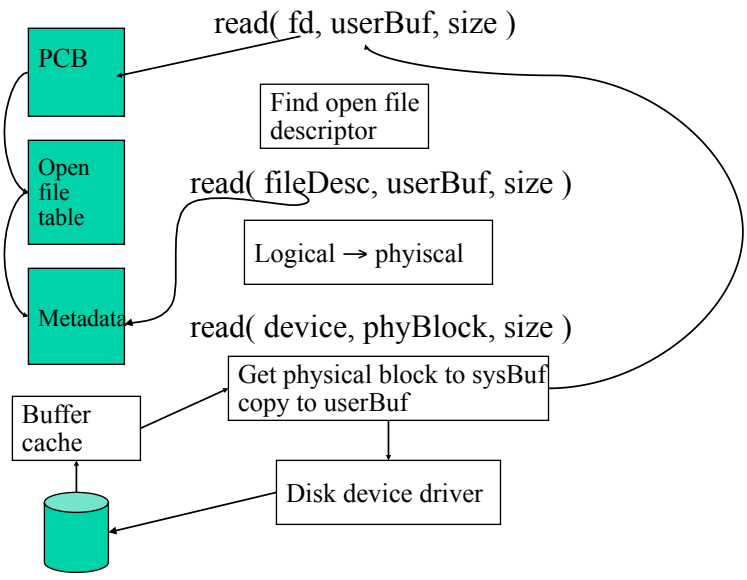
1. Check if valid call
2. Allocate file descriptor
3. If file exists, open for read. Otherwise, create a new file. Must get directory i-node. May require disk I/O.
4. Set access rights, flags and pointer to v-node
5. Return index to file descriptor table

fd

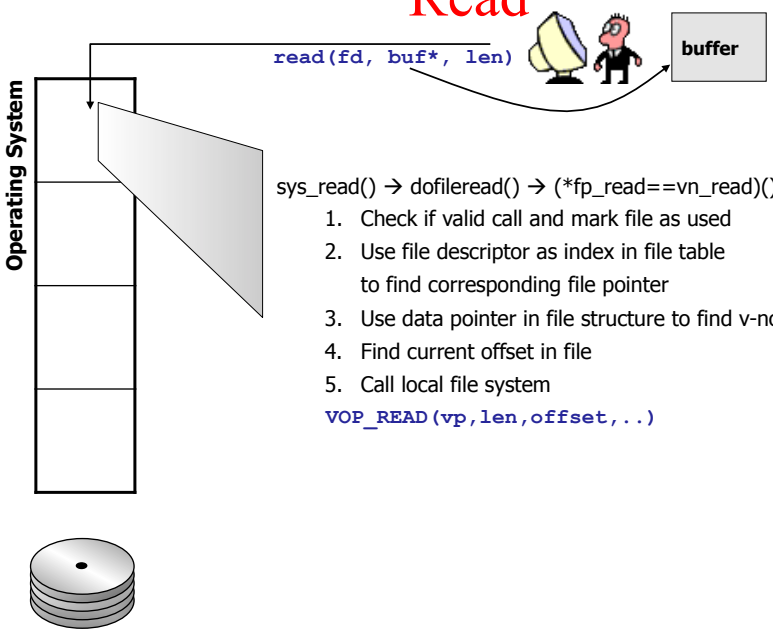
From User to System View

- What happens if user wants to read 10 bytes from a file starting at byte 2?
  - seek byte 2
  - fetch the block
  - read 10 bytes
- What happens if user wants to write 10 bytes to a file starting at byte 2?
  - seek byte 2
  - fetch the block
  - write 10 bytes
  - write out the block

Read Block



Read

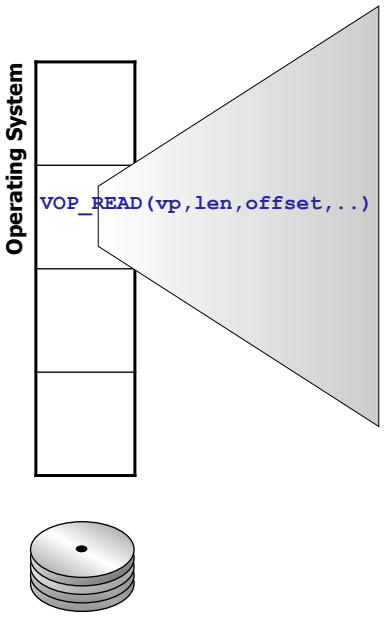


sys_read() → dofileread() → (*fp_read==vn_read)():

1. Check if valid call and mark file as used
2. Use file descriptor as index in file table to find corresponding file pointer
3. Use data pointer in file structure to find v-node
4. Find current offset in file
5. Call local file system

```
VOP_READ(vp,len,offset,..)
```

Read



VOP_READ(...) is a pointer to a read function in the corresponding file system, e.g., Fast File System (FFS)

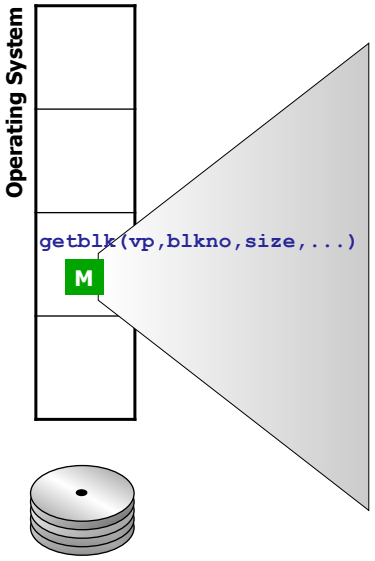
READ():

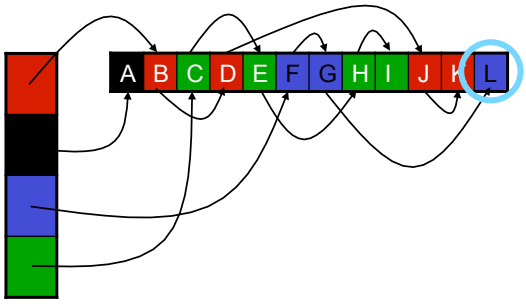
1. Find corresponding inode
2. Check if valid call - file size vs. len + offset
3. Loop and find corresponding blocks
   - find logical blocks from inode, offset, length
   - do block I/O, fill buffer structure
     e.g., bread(...) → bio_doread(...) → getblk()

     ```
     getblk(vp,blkno,size,...)
     ```

   - return and copy block to user

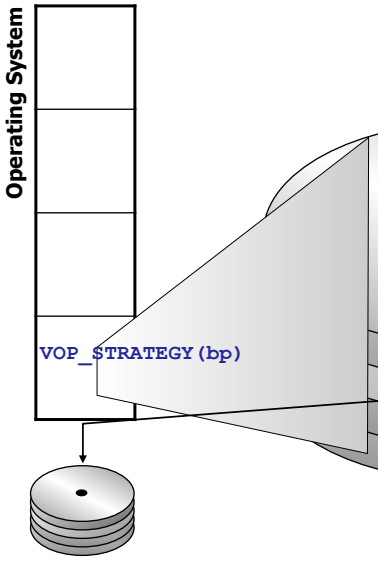
Read





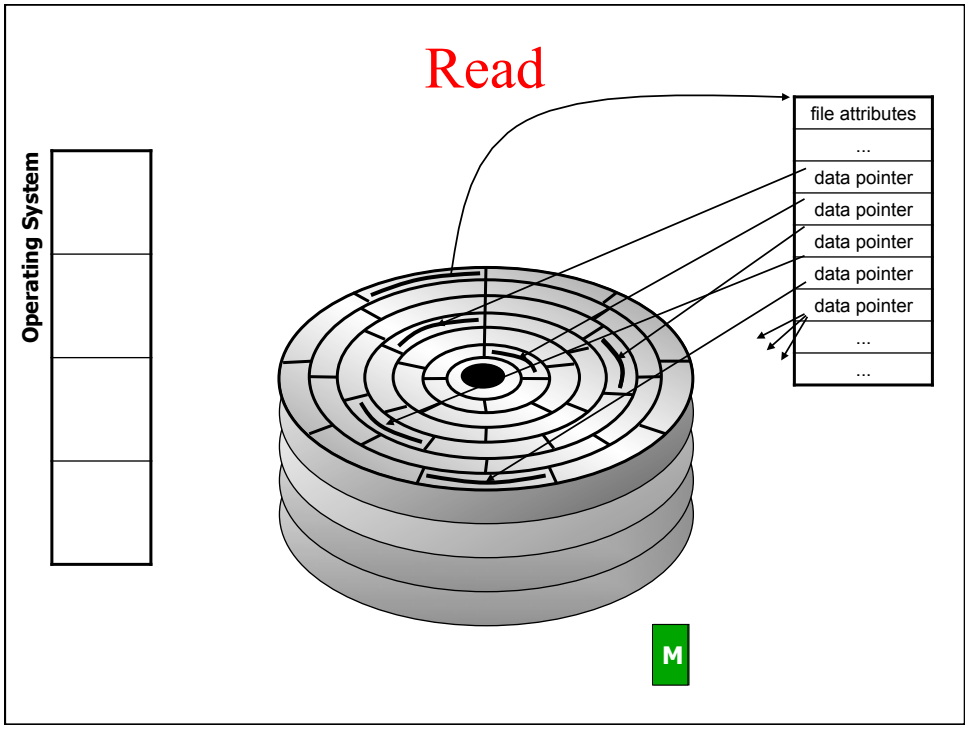
1. Search for block in buffer cache, return if found (hash vp and blkno and follow linked hash list)
2. Get a new buffer (LRU, age)
3. Call disk driver - sleep or do something else
   `VOP_STRATEGY(bp)`
4. Return buffer

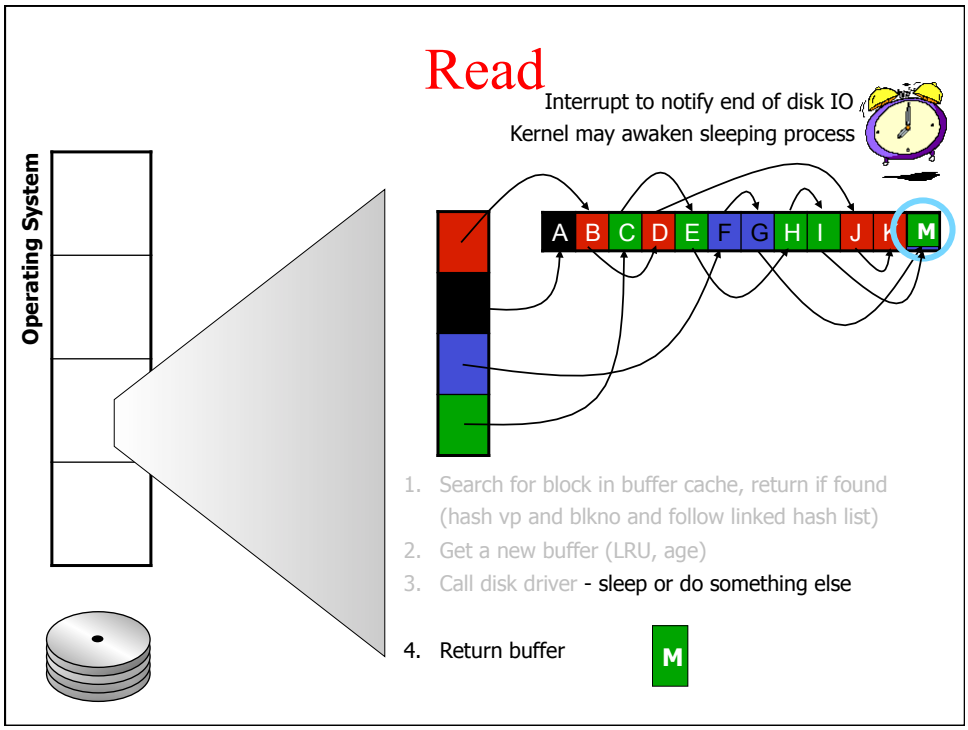
Read



VOP_STRATEGY(...) is a pointer to the corresponding driver depending on the hardware,
e.g., SCSI - sdstrategy(...) → sdstart(...)

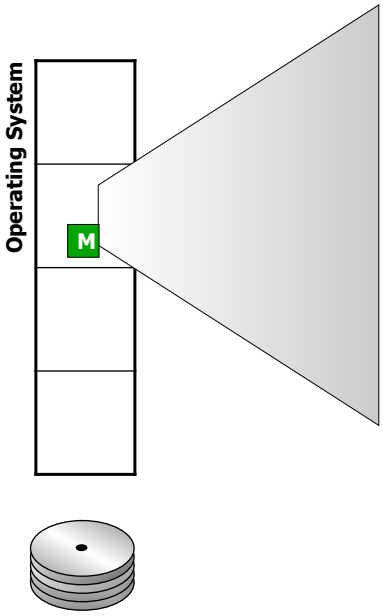
1. Check buffer parameters, size, blocks, etc.
2. Convert to raw block numbers
3. Sort requests according to SCAN - disksort_blkno(...)
4. Start device and send request


Read
Operating System
file attributes
...
data pointer
data pointer
data pointer
data pointer
data pointer
...
...
M


Read
Interrupt to notify end of disk IO
Kernel may awaken sleeping process
Operating System
A B C D E F G H I J K M
1. Search for block in buffer cache, return if found
(hash vp and blkno and follow linked hash list)
2. Get a new buffer (LRU, age)
3. Call disk driver - sleep or do something else
4. Return buffer
M

Read

buffer

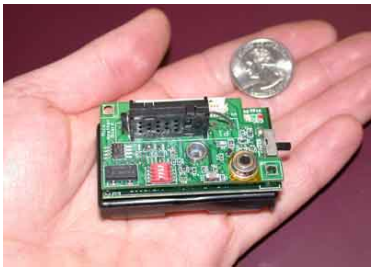


READ():

1. Find corresponding inode
2. Check if valid call - file size vs. len + offset
3. Loop and find corresponding blocks
   - find logical blocks from inode, offset, length
   - do block I/O,
     e.g., bread(...) → bio_doread(...) → getblk()

- return and copy block to user

Outlook to Research Issues



58

Outlook to Research Issues (cont.)

- Energy aware OS
- Impact of memory speed on performance
- Video streaming on small devices
- Sensor data processing on very small devices

59

Concluding Questions

- What is the basic layout of FS on disk?
- What is the relationship between a directory and an i-node?
- How can we manage free blocks?
- What is the difference between i-node and v-node?

60