EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

File systems

File Systems

Essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. Information must survive termination of process using it.
3. Multiple processes must be able to access information concurrently.

File Systems

Think of a disk as a linear sequence of fixed-size blocks and supporting two operations:

1. Read block *k*.
2. Write block *k*

The file system is that thing that makes this useful

Similar to Memory Management

The abstraction of files, directories, links, permissions, ... - corresponds to virtual memory management

Keeping track of disk blocks, placement, ... : corresponds to physical memory management

Metadata: The file system information used to keep track of the data : corresponds to page table, segmentation, regions, mappings...

What we will cover

- The abstractions of File systems:
  - what is a file, naming/directories
- Physical management:
  - disk (storage device) layout
  - blocks used in a file
  - free blocks
- Integrated into the OS - VFS
- Dealing with failure and maintaining metadata consistency

The abstraction: 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

File Naming

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

Figure 4-1. Some typical file extensions.

File Structure

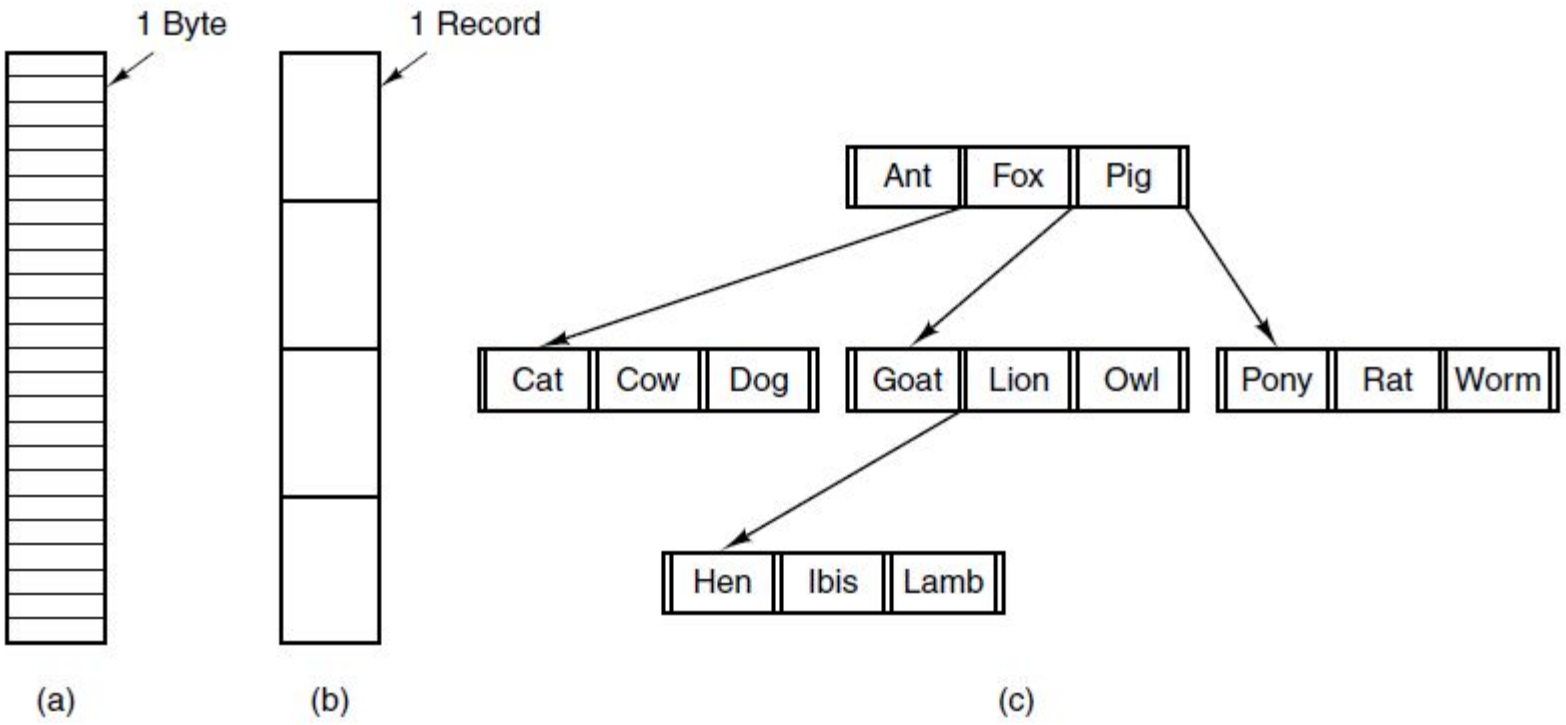


Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

File Structure

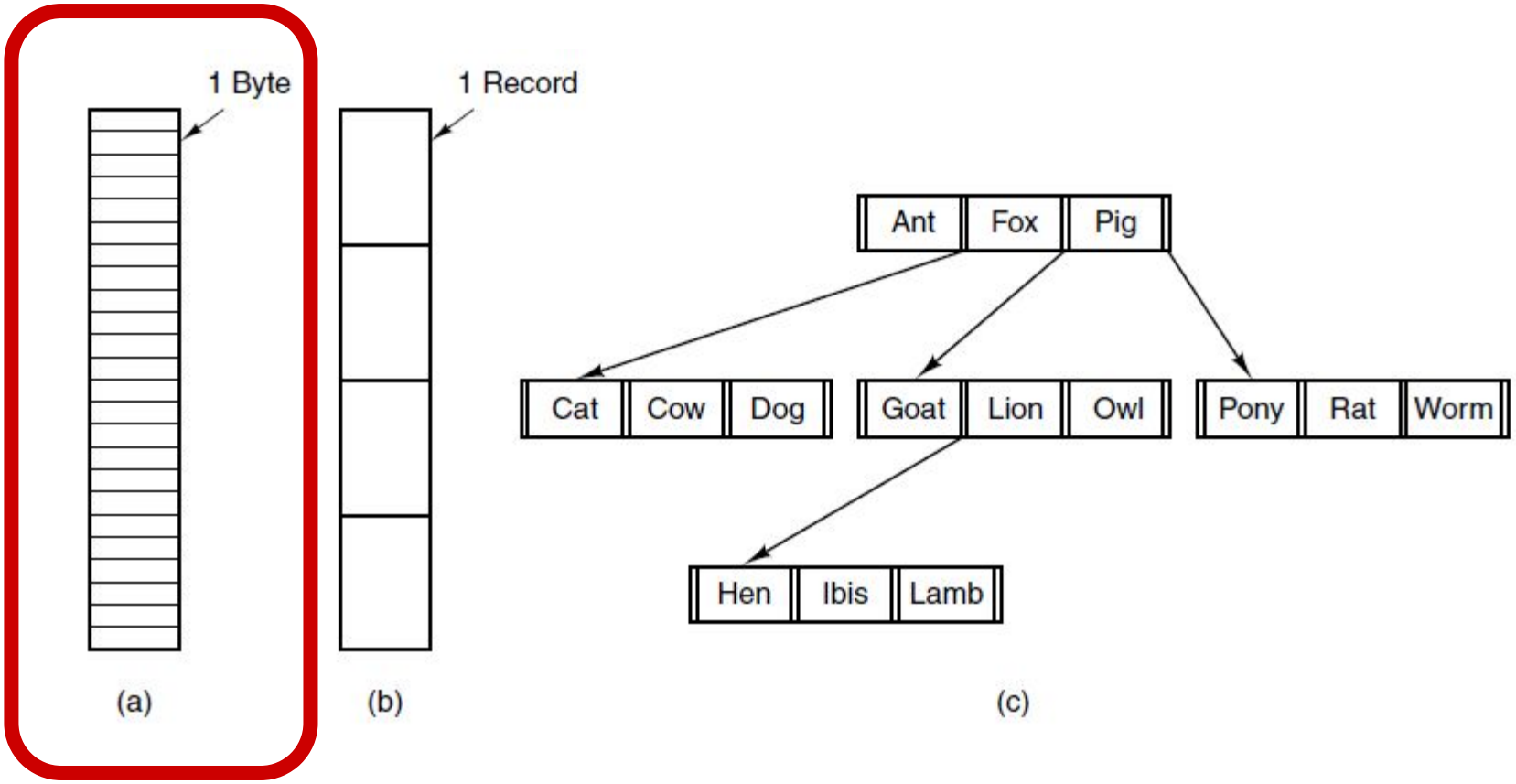


Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Binary File Types

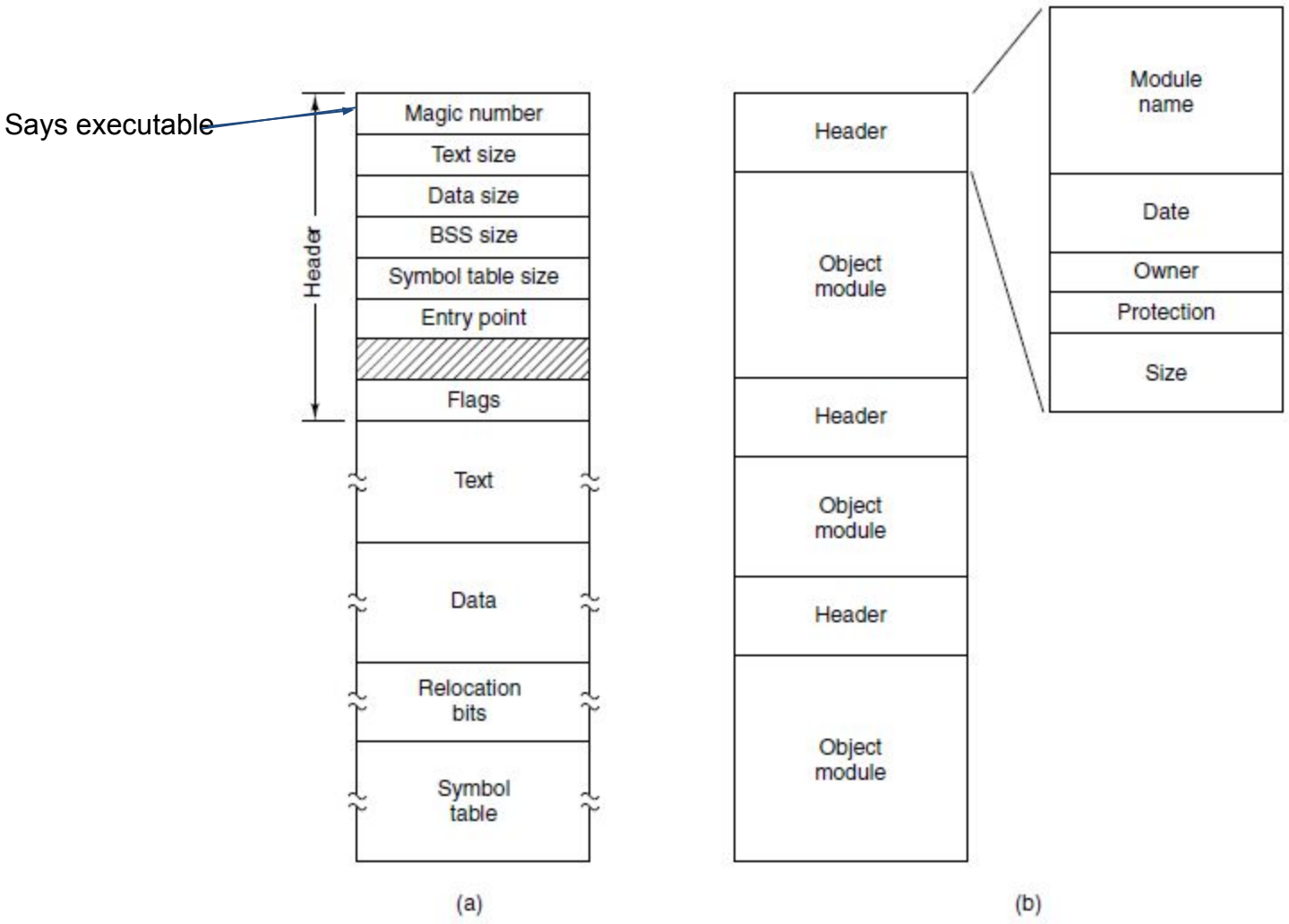


Figure 4-3. (a) An executable file. (b) An archive

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 was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 4-4. Some possible file attributes.

Linux “stat” structure

```
struct stat {
            dev_t     st_dev;         /* ID of device containing file */
            ino_t     st_ino;         /* Inode number */
            mode_t    st_mode;        /* File type and mode */ - RWX for usr/group/all
            nlink_t   st_nlink;       /* Number of hard links */
            uid_t     st_uid;         /* User ID of owner */
            gid_t     st_gid;         /* Group ID of owner */
            dev_t     st_rdev;        /* Device ID (if special file) */
            off_t     st_size;        /* Total size, in bytes */
            blksize_t st_blksize;     /* Block size for filesystem I/O */
            blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
            struct timespec st_atim;  /* Time of last access */
            struct timespec st_mtim;  /* Time of last modification */
            struct timespec st_ctim;  /* Time of last status change */
          };
```

e.g. used make

Types of files in unix

- Regular files - ascii or binary (FS doesn't care)
- Directories
- Character files - serial I/O devices
- Block special - disks

Using file system Demo

Example Program Using File System Calls (1)

```
/* 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 */

    /* Open the input file and create the output file */
```

Figure 4-5. A simple program to copy a file.

Example Program Using File System Calls (2)

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

/* 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 */
```

Figure 4-5. A simple program to copy a file.

Example Program Using File System Calls (3)

```
        /* 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 */
}
```

Figure 4-5. A simple program to copy a file.

Single-Level Directory Systems

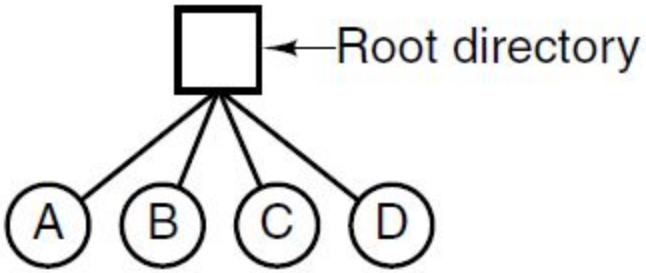


Figure 4-6. A single-level directory system containing four files.

Hierarchical Directory Systems

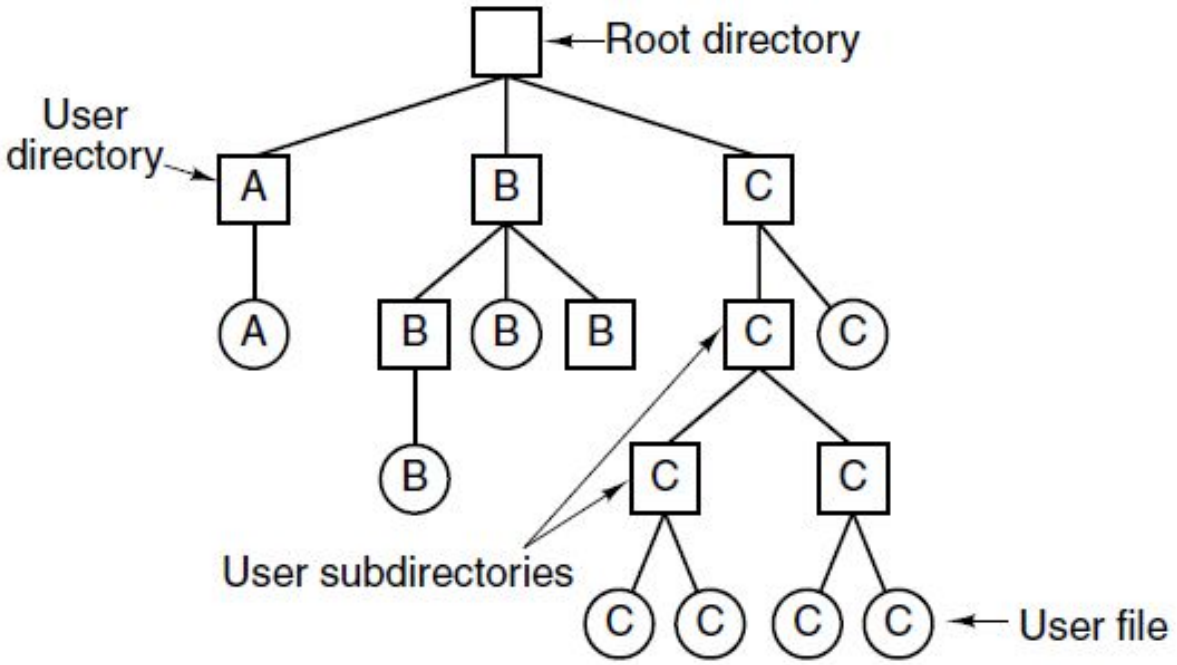
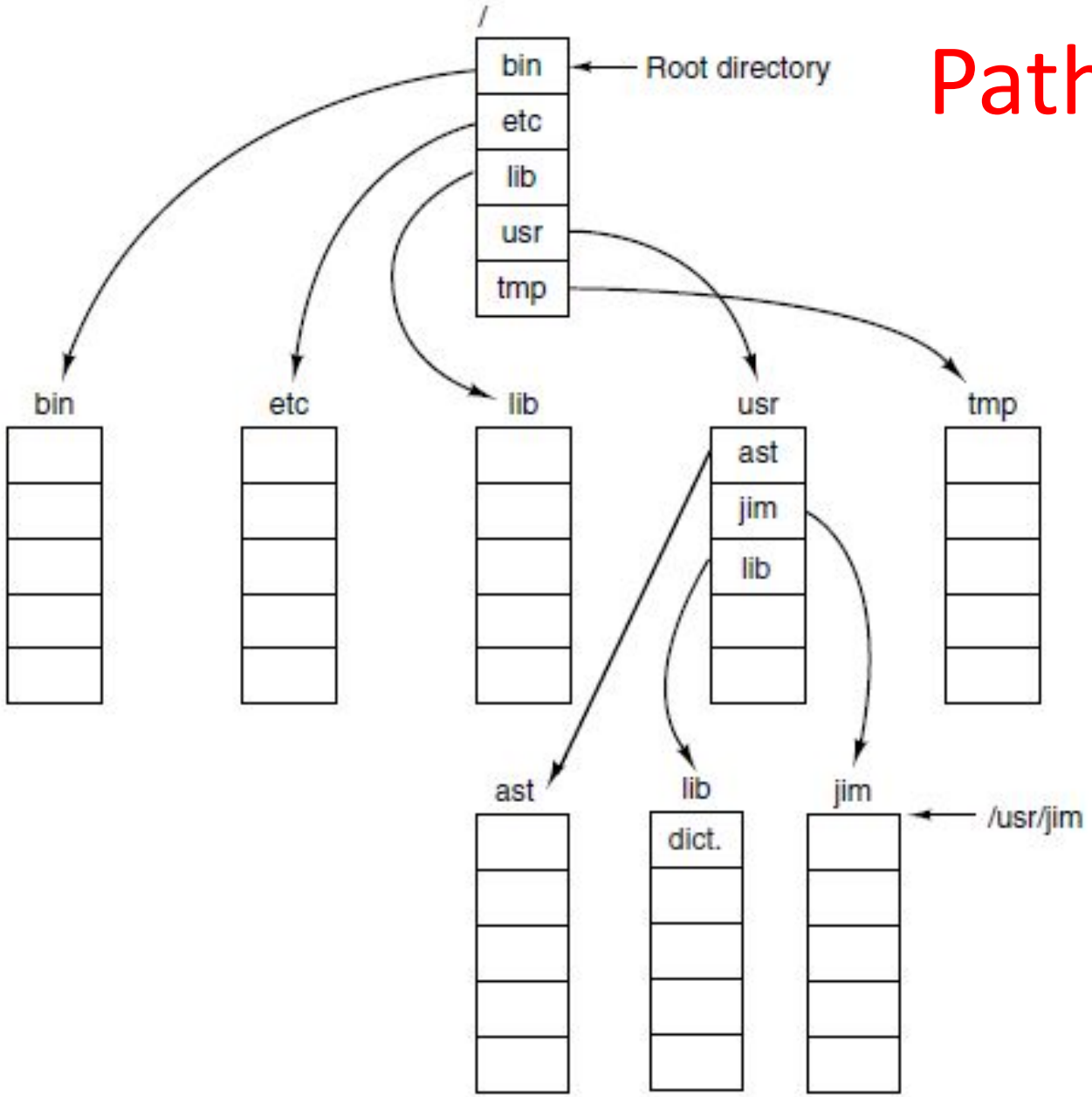


Figure 4-7. A hierarchical directory system.

Path Names



Referring to files

- Absolute path, e.g., /usr/jim/foo/bar
- relative path foo/bar (cwd is /usr/jim)
- ".." refers to parent directory
  - e.g., ../foo/bar (cwd is /usr/jim/src)

Directory Operations

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

What we will cover

- The abstractions of File systems:
  - what is a file, naming/directories
- Physical management:
  - disk (storage device) layout
  - blocks used in a file
  - free blocks
- Integrated into the OS - VFS
- Dealing with failure and maintaining metadata consistency

Disk Layout

- MBR - Master boot record
- Partition table
  - Start/end of each partition
  - Marker for active partition used to boot OS
- Contents of partition is file system specific

Disk Layout

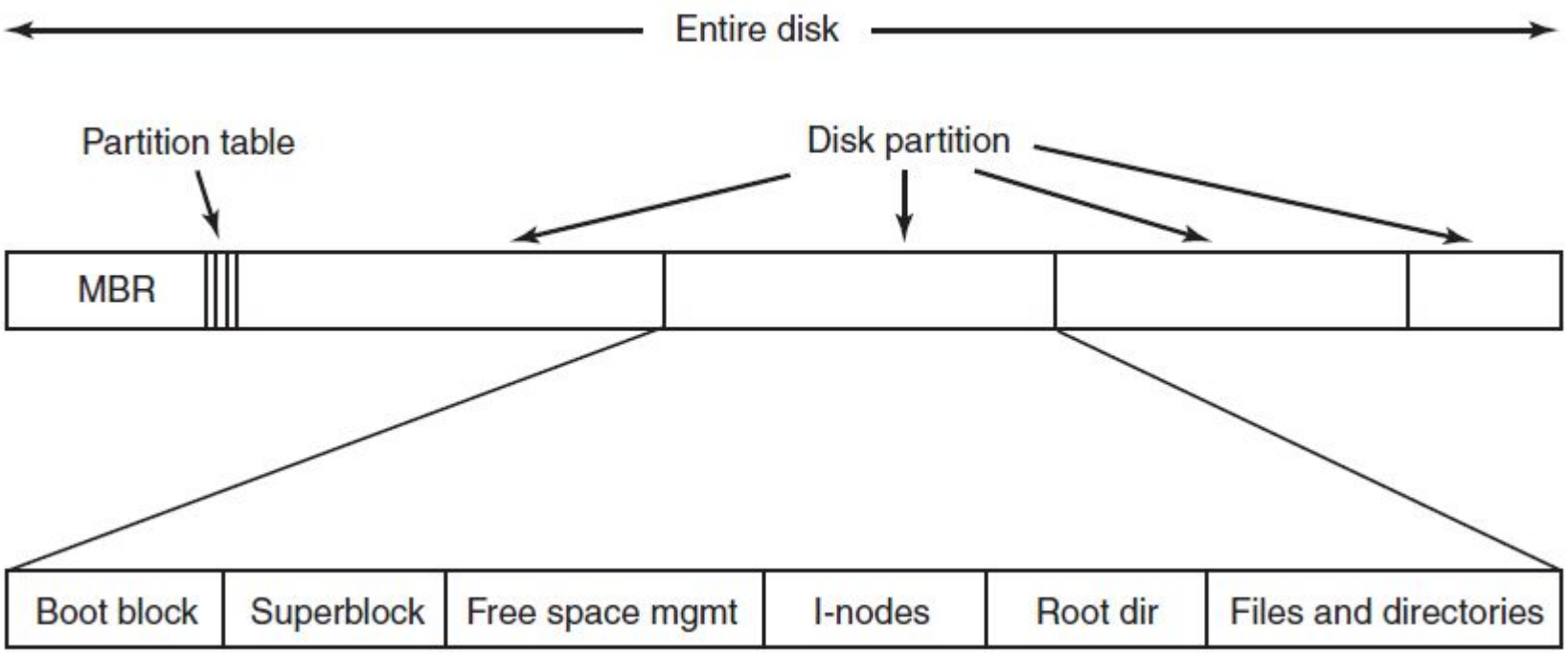


Figure 4-9. A possible file system layout.

Implementing Files
Contiguous Layout

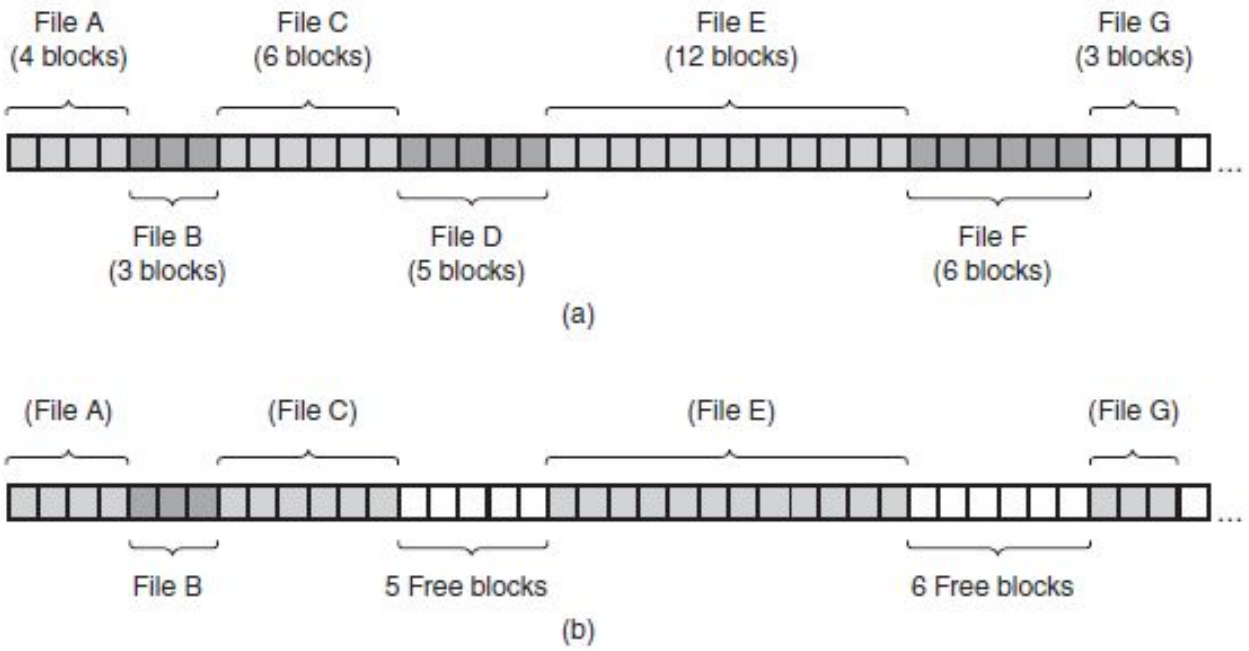


Figure 4-10. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

Trade-offs

- Contiguous layout advantages
  - very dense meta-data
  - very fast read - one seek operation
- Disadvantages
  - rapidly fragments disks - external fragmentation
  - can’t extend files
- Used CD-ROMs
- Varient extent-based file systems

Implementing Files
Linked List Allocation

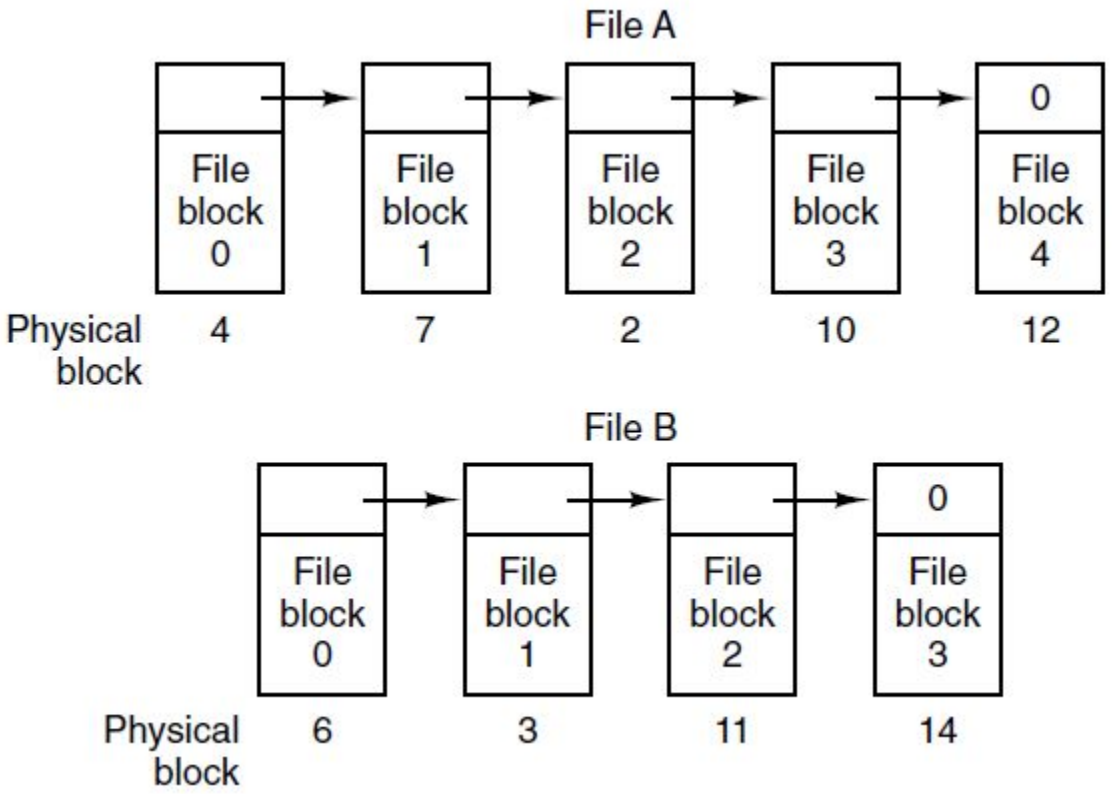


Figure 4-11. Storing a file as a linked list of disk blocks.

Trade-offs

- No external fragmentation
- Directory just stores block of start of file.
- Disadvantages
  - block size no longer power of 2 - link in the block...
  - need to read sequentially

Implementing Files
Linked List – Table in Memory

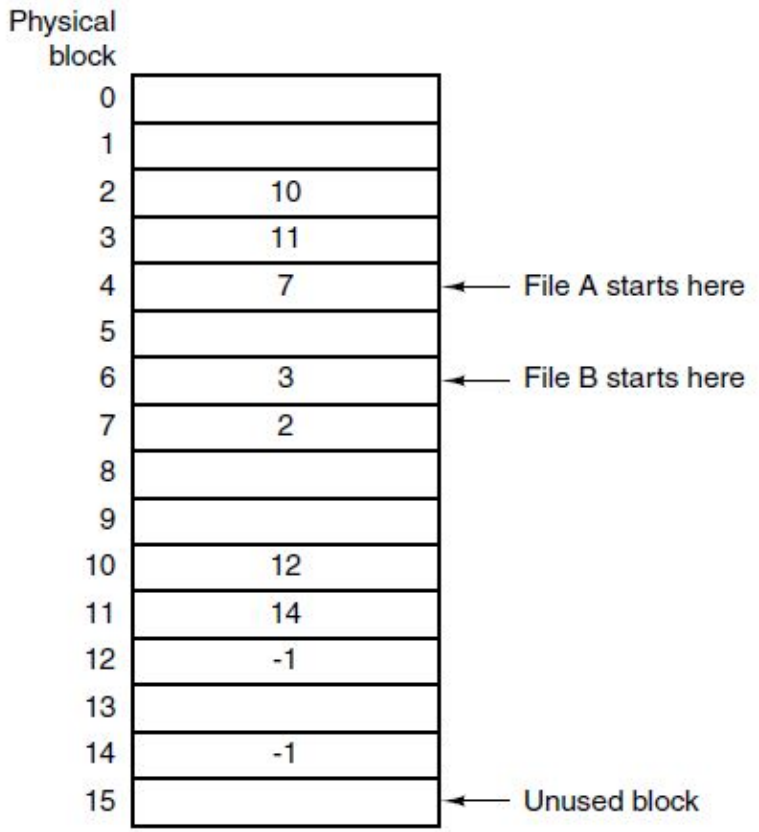


Figure 4-12. Linked list allocation using a file allocation table in main memory.

Trade-offs

- Still no external fragmentation
- Solves the two disadvantages of previous scheme:
  - no link in the block...
  - avoids need to read sequentially
- Problem: in memory table huge, for !TB disk, with 1KB block size, need 1 billion entries of at least 3bytes, i.e., 3BG of memory
- Scheme for original MS-DOS FS

Index nodes or i-nodes

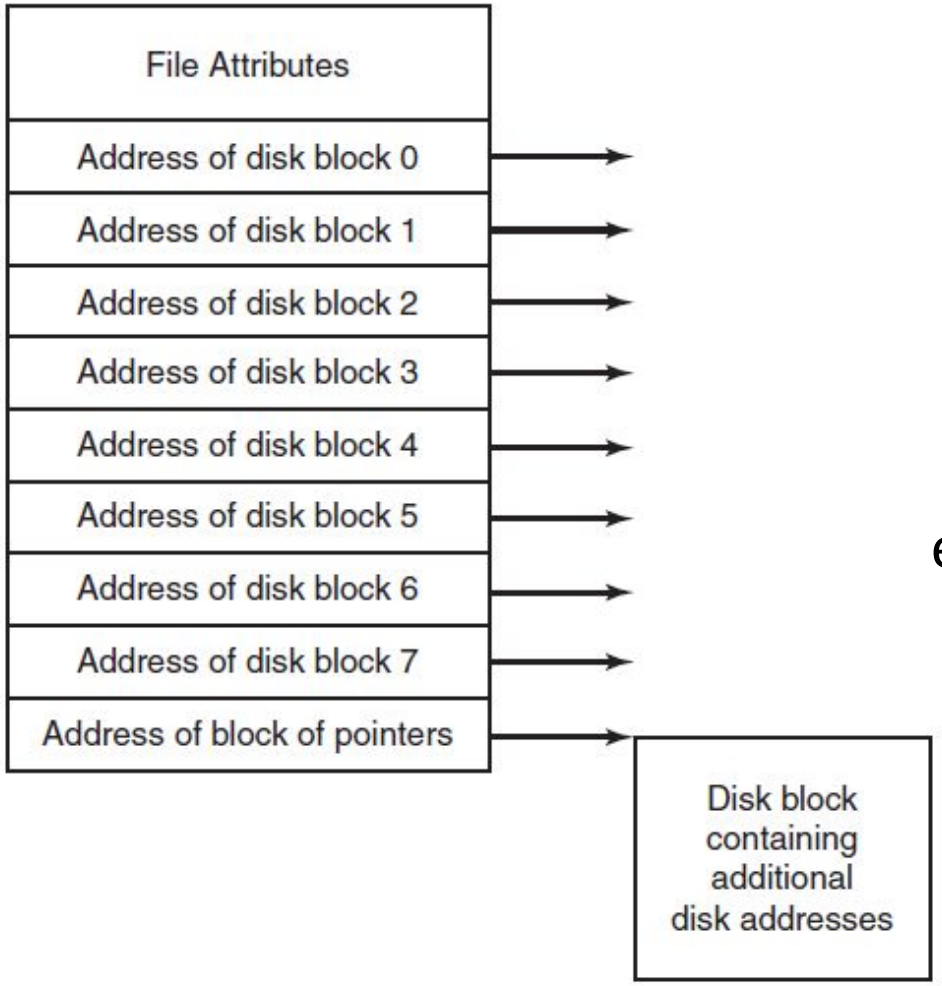
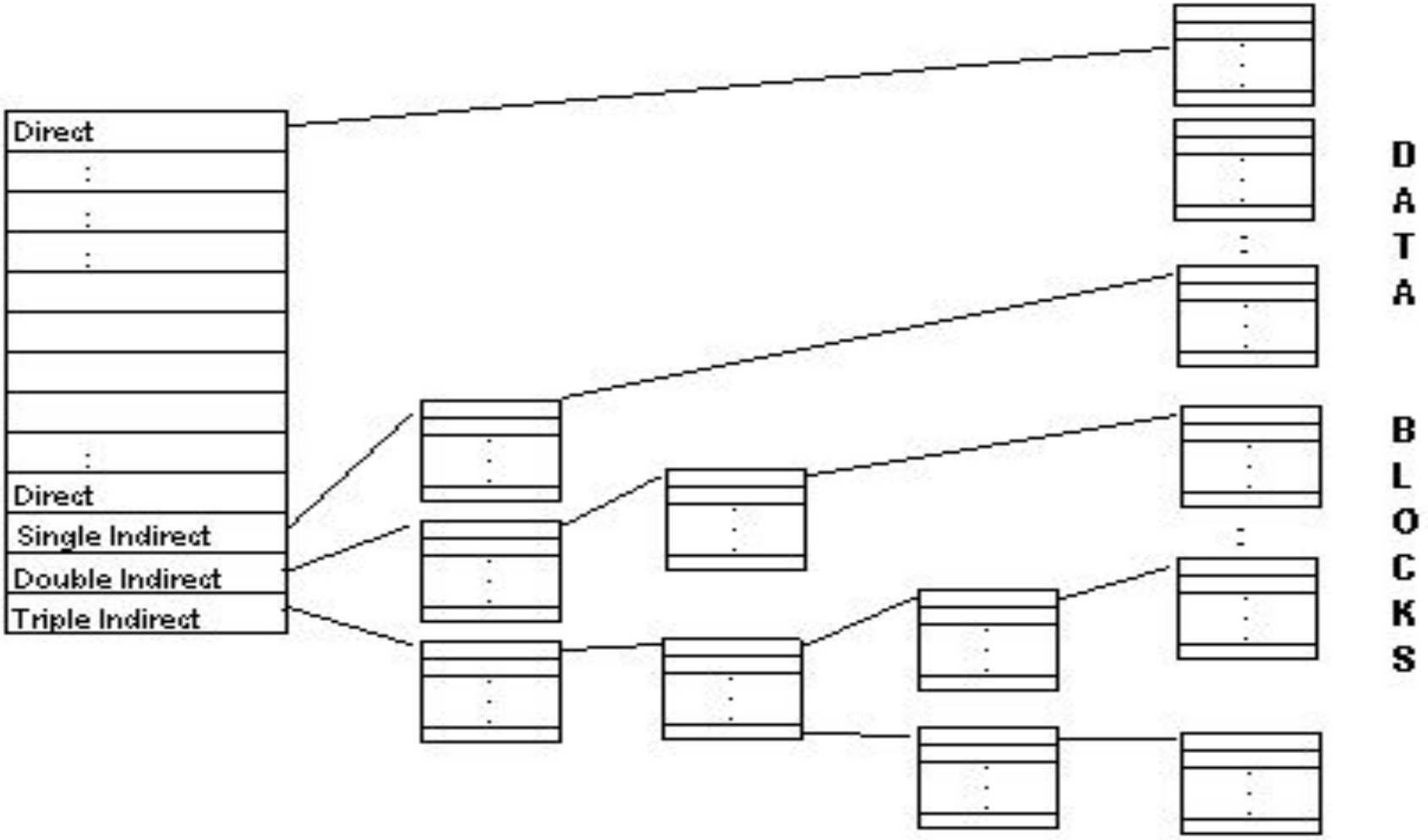
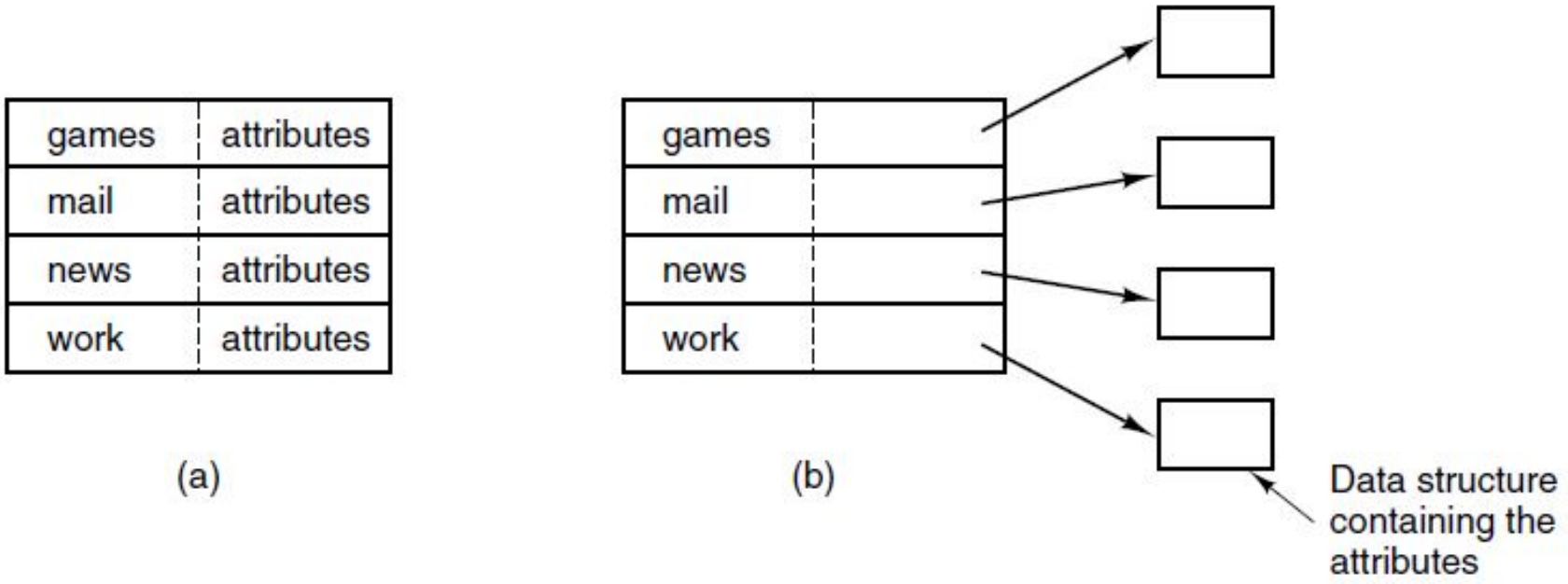


Figure 4-13. An example i-node.

To handle larger files

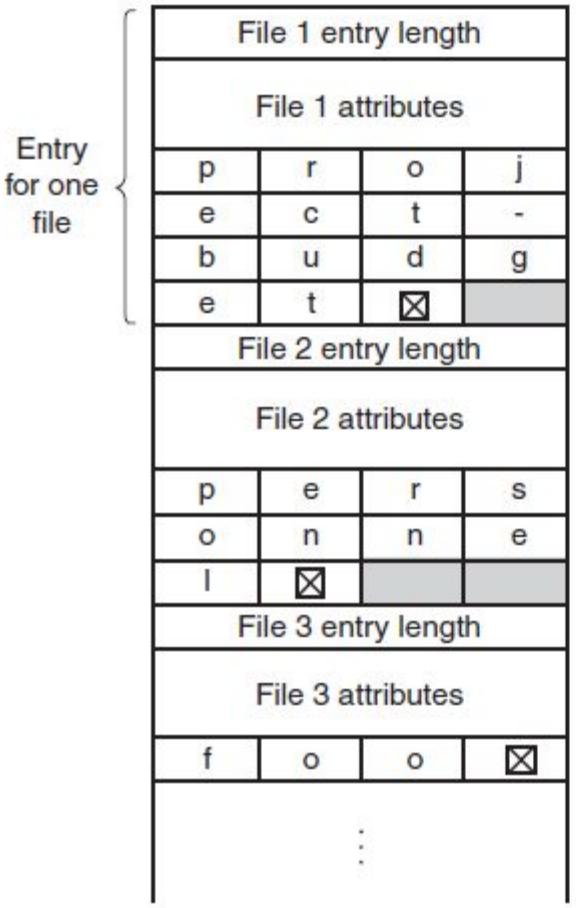


Implementing Directories (1)



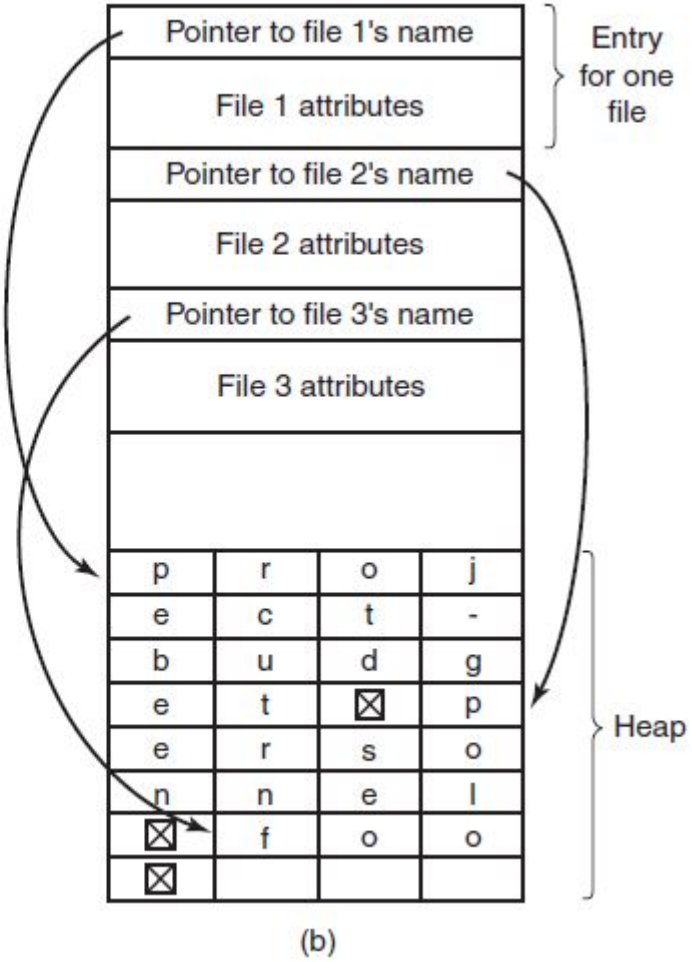
(a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry.
(b) A directory in which each entry just refers to an i-node.

Implementing Directories in-line



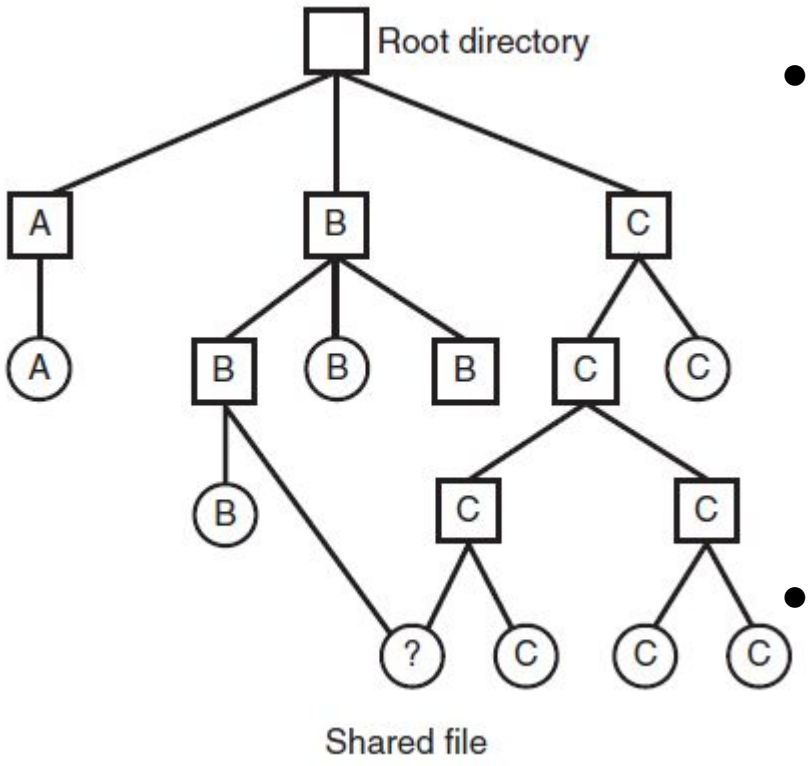
- Historically directory entries are variable length
- What about file names 1000s characters
- Disadvantage?

Implementing Directories (2)



- Lots of other alternatives:
  - file names at the end
  - hash table in secondary data structure
- IMHO, probably not worth it:
  - more efficient on cache to pack
  - most file names small
  - most directories small
  - can re-organize in memory
  - ….
  - but sometimes for massive file names and massive directories performance is bad
  - OSes are compromises; but there are specialized FS, e.g., HPC

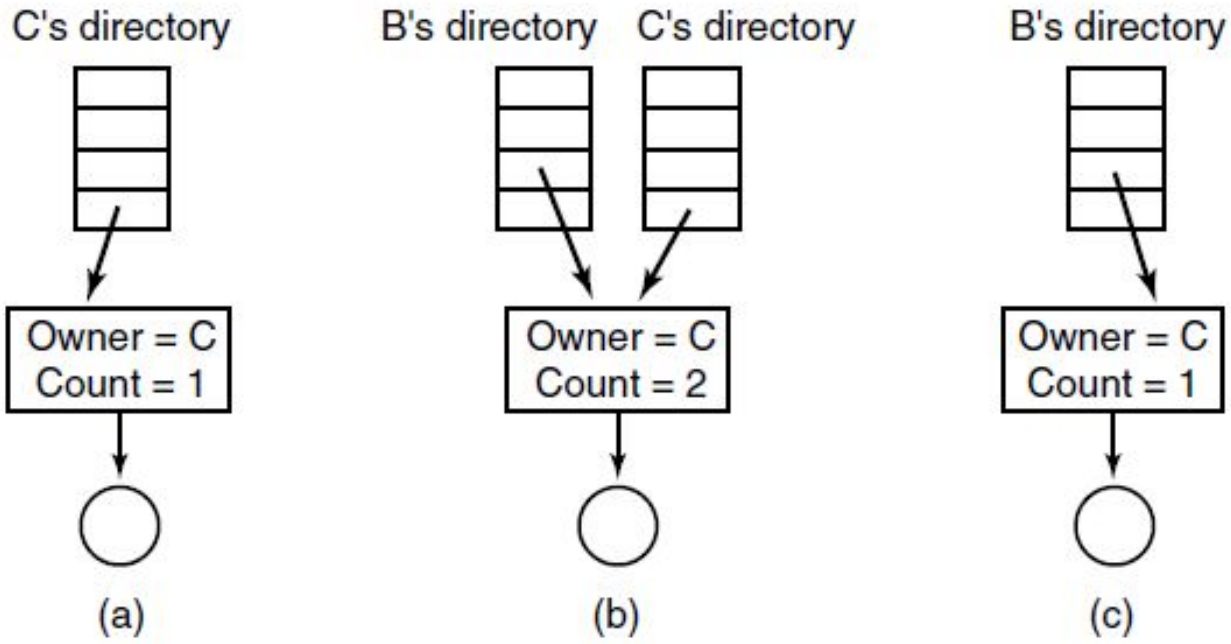
Shared Files



Shared file

- Super useful to reference same file in multiple places. Why?
- Can share files in directory in two ways:
  a. softlinks - there is one owner, other directory just contains a string that references it (special inode)
  b. hardlink - two directory entries point to the same inode
- softlinks require traversing file system paths on each access

Shared Files hard links



- Hardlinks require counter in inode
- C continues to be billed after unlink

Disk space management

Some statistics

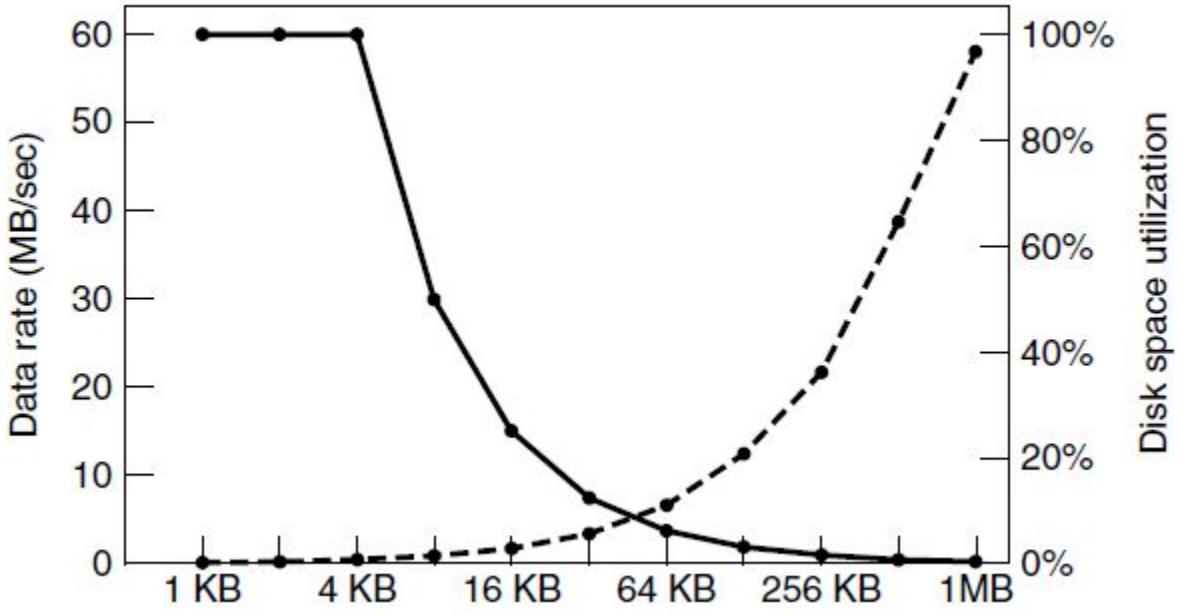
Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69

Length	VU 1984	VU 2005	Web
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

What we see

- With block size of 4KB 60-70% of files fit in a single block
- From other data, vast majority of disk space used by a modest number of large files
- Hence, we don’t care about internal fragmentation

Tradeoff disk space vs performance

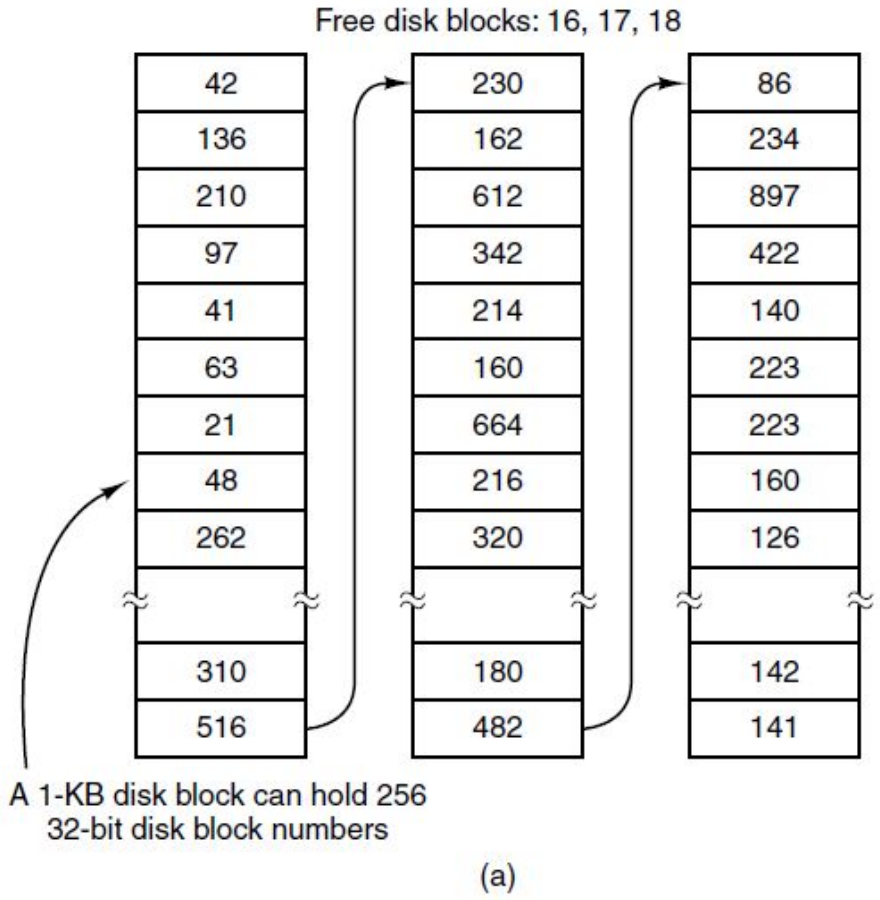


The dashed curve (left-hand scale) gives the data rate of a disk. The solid curve (right-hand scale) gives the disk space efficiency. All files are 4 KB.

Block size

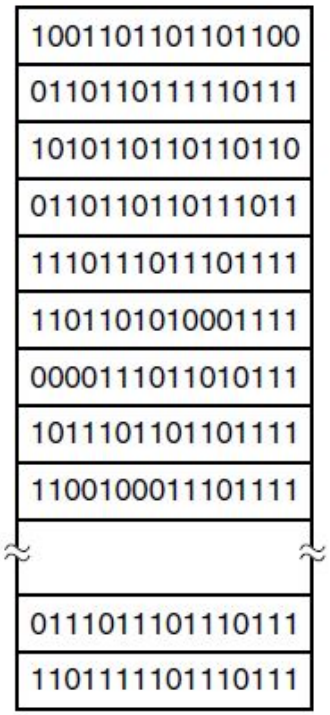
- Historically FS have chosen block sizes in the range of 1K-4K
- 4K is nice, since matches minimum page size, modest space overhead
- Disk meta data… increase with smaller block size
- With massive size of current disks, perhaps time to reconsider

Keeping track of free blocks using linked list



- Chained through free blocks
- For 1TB disk, this means 1M blocks to hold free list
- Huge initialization cost
- Can maintain ranges, rather than individual blocks
- Compresses to small number of blocks when disks full

Keeping track of free blocks



A bitmap

(b)

- 1 bit per block, for 1TB disk, this means 130K blocks to hold free inf
- Still large initialization cost, but easier to batch (e.g., write all zeroes)
- Easier to find contiguous regions of disk
- Can allocate from one block at a time, keeping disk head in one region disk
- When disk fragmented, each block may have just few bits available

Discussion

- There is no right answer, OS are a compromise between many different applications needs
- The more we know the expected access pattern, the better job we can do.
- Different file systems can/are optimized for different usage, but…
  - often time we have a wide diversity of applications using data
  - can only partition your disk so far...

Virtual File System layer

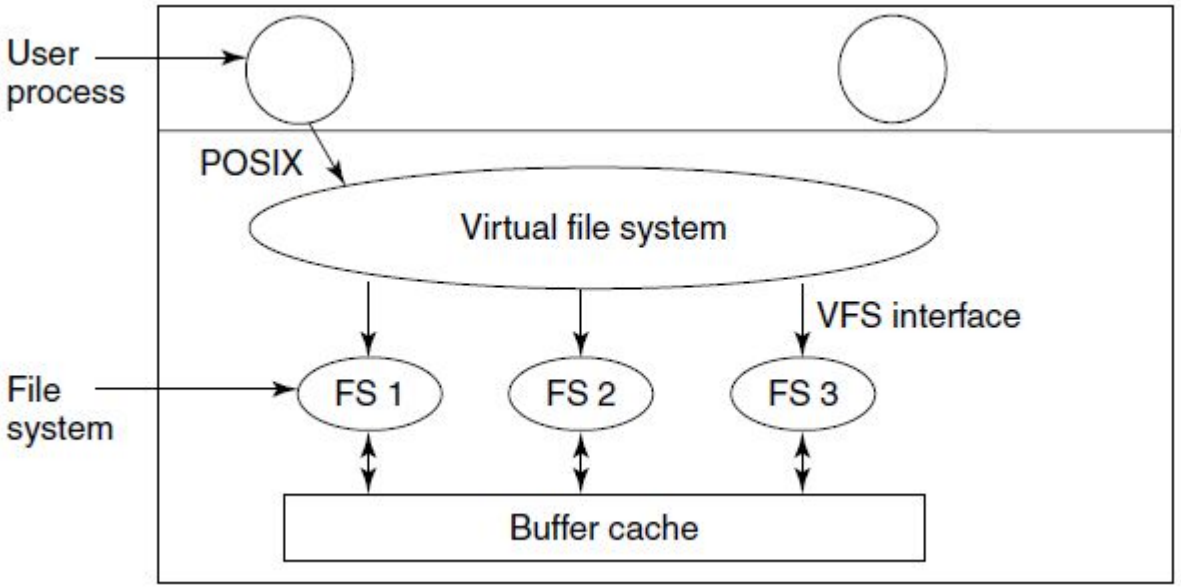
What we will cover

- The abstractions of File systems:
  - what is a file, naming/directories
- Physical management:
  - disk (storage device) layout
  - blocks used in a file
  - free blocks
- Integrated into the OS - VFS
- Dealing with failure and maintaining metadata consistency

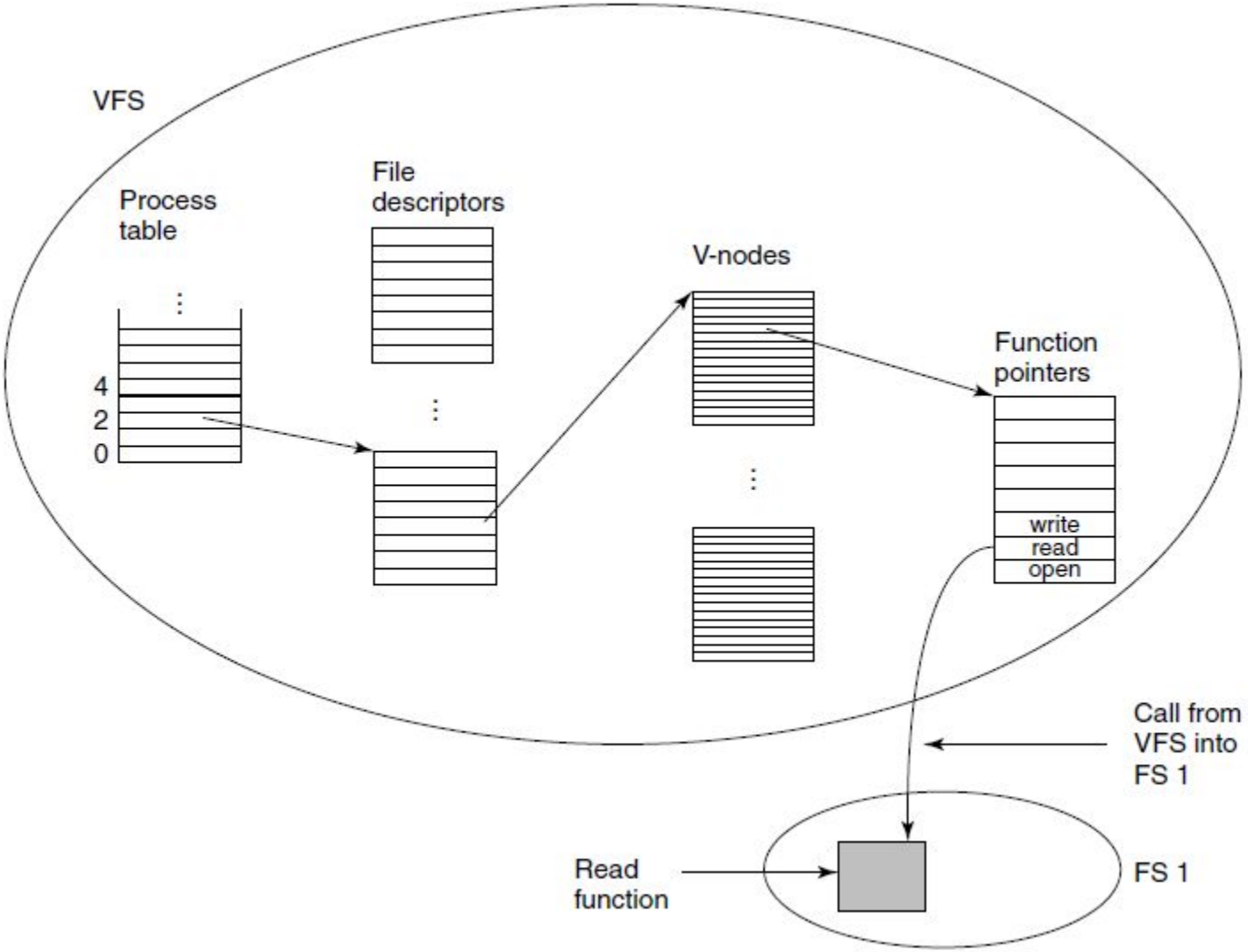
VFS layer

- The operating system must support potentially multiple concurrent file systems:
  - NFS, DVD FS, Disk file system…
- The VFS implements all FS calls, and translates them in an OO fashion to calls on:
  - superblocks, inodes, dentries, files, address space
- The file system, in turn, makes calls to the underlying buffer cache to cache its data, and the devices
- All file systems mounted into a single name space.

Virtual File Systems (1)



Putting it together



Putting it together

- Load a file system module
- File system loads superblock from disk
- All operations to open files/directories relative to that, load inode into memory along with associated file pointers
- See here for full description on how a file system interacts with the VFS

Example Linux SB

Global operations on a file system

```
struct super_operations {
      struct inode *(*alloc_inode)(struct super_block *sb);
      void (*destroy_inode)(struct inode *);
      void (*put_super) (struct super_block *);
      int (*sync_fs)(struct super_block *sb, int wait);
      int (*freeze_fs) (struct super_block *);
```

```
      void (*umount_begin) (struct super_block *);
      int (*show_options)(struct seq_file *, struct dentry *);
      ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
      ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
      int (*nr_cached_objects)(struct super_block *);
     void (*free_cached_objects)(struct super_block *, int);
};
```

Operations on open files

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
```

```
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned
long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset,
                          loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
};
```

Example Linux inode operations

```
struct inode_operations {
        int (*create) (struct inode *,struct dentry *, umode_t, bool);
        struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
```

```
        const char *(*get_link) (struct dentry *, struct inode *,  struct delayed_call *);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (const struct path *, struct kstat *, u32, unsigned int);
```

Most FS will have different operations for directories and files

```
        int (*atomic_open)(struct inode *, struct dentry *, struct file *,
                           unsigned open_flag, umode_t create_mode);
        int (*tmpfile) (struct inode *, struct dentry *, umode_t);
};
```

Address space

```
struct address_space_operations {
        int (*writepage)(struct page *page, struct writeback_control *wbc);
        int (*readpage)(struct file *, struct page *);
        int (*writepages)(struct address_space *, struct writeback_control *);
        int (*set_page_dirty)(struct page *page);
        int (*readpages)(struct file *filp, struct address_space *mapping,
```

```
        int (*launder_page) (struct page *);

        int (*is_partially_uptodate) (struct page *, unsigned long,
                                      unsigned long);
        void (*is_dirty_writeback) (struct page *, bool *, bool *);
        int (*error_remove_page) (struct mapping *mapping, struct page *page);
        int (*swap_activate)(struct file *);
        int (*swap_deactivate)(struct file *);
};
```

What we will cover

- The abstractions of File systems:
  - what is a file, naming/directories
- Physical management:
  - disk (storage device) layout
  - blocks used in a file
  - free blocks
- Integrated into the OS - VFS
- Dealing with failure and maintaining metadata consistency

FS Metadata consistency

Consider removing a file

1. Remove file from its directory.
2. Release i-node to the pool of free i-nodes.
3. Return all disk blocks to pool of free disk blocks

But… everything cached… what could go wrong?.

What could go wrong

Imagine if system crashes:

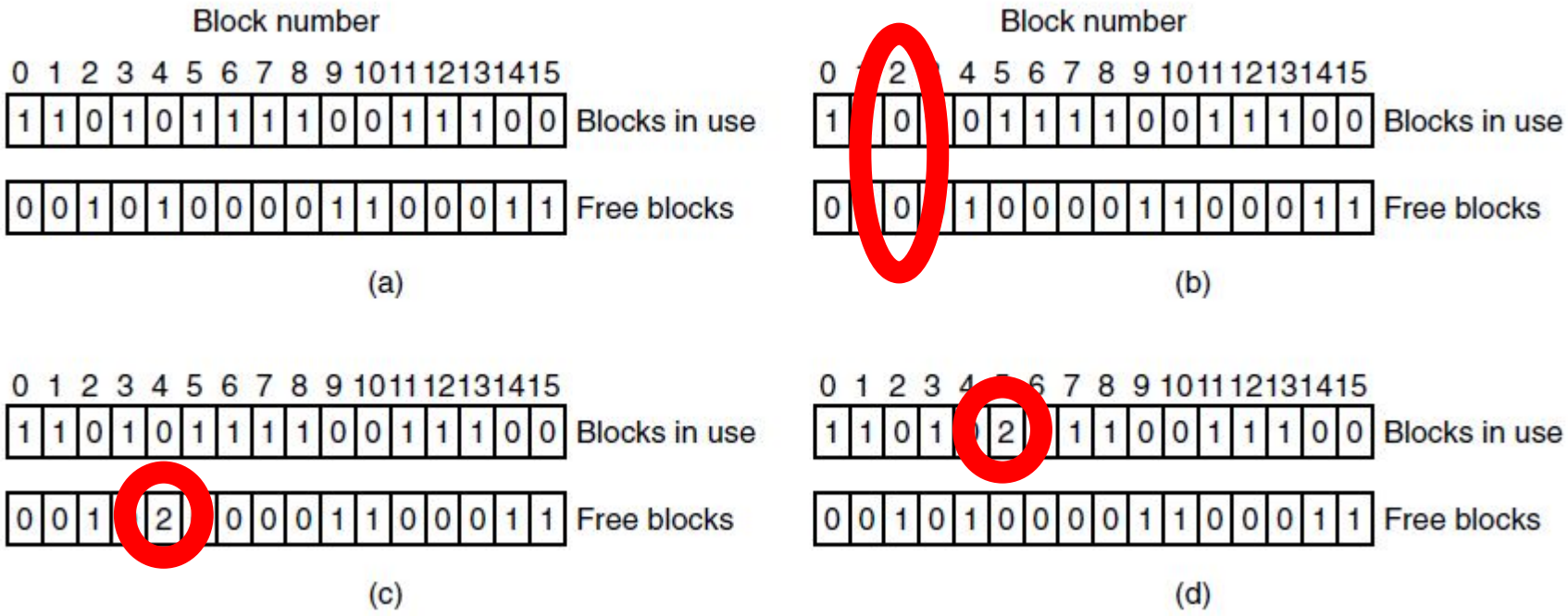
1. After freeing inode but before freeing disk blocks
2. After freeing disk blocks, before freeing inode
3. Similar problems on writing ….

Original approach - fsck

fsck - walk through file inodes, check that:

1. each block either free or used in some file once
2. all inodes in a directory routed from top
3. all ref counts in inodes match directory refs
4. all freed disk blocks not referred to by file
5. …

e.g. counter list used/free



This is expensive, can take many hours, system could crash again.

Failures happen

- We need to accept that data will be lost unless we implement a write through cache and block process.
- Sometimes we will get disk errors… so fsck will always be needed, but...
- Can we get have consistent on-disk state not have any corruption from system crashes/power failures?
  - (some orphaned state may be okay for a while)

Approach 1: synchronous ordered writes meta-data

Logically order writes to preserve dependencies & sync write to disk, e.g.:

a. don't free blocks until released inode
b. create allocates inode before directory entry

This can be incredibly expensive, nobody does it anymore.

You could throw HW at the problem; e.g. NVRAM

Approach 2: soft update

1. Maintain dependency of meta-data in memory
2. Data blocks can be written at any time.
3. When meta-data is being written, done asynchronously but ordered:
   a. e.g., inode must reach disk before directory

Disk is in a crash consistent state, data may be lost, but meta-data is consistent (mostly)

Approach 3: Journaling/Log

Technique originally used for DB:

- log separate from FS metadata with an entry for every change
- if failure occurs, re-play log of complete operations; abort partial ones

Great comparison here; journaling won, don't really know why.

Key is that logged operations idempotent, i.e.can repeat without harm. Is free block A idempotent if:

1. you have a linked list of free blocks NO
2. you have a bit vector of free blocks YES

Log-structured FS

The best research is based on hypothesis.

Imagine if the memory huge? With large file caches, how would this change the operations that get to disk?

FS are designed to optimize for reads (e.g., extent based)

Log-structured FS idea

- Organize entire disk into a massive log
- Write out segments, with inodes, directories, blocks as a big (e.g., 1MB) write.
- Maintain map of inodes to location in last segment.
- Garbage collect segments with small amount of data

Trade offs

- The good:
  - Recovery can be very fast.
  - Can easily integrate snapshots
  - Writes are huge, very efficient on disk
- The bad:
  - Reads can be slower, data may be scattered over disk; but most files written at the same time.
- Is now used in SSDs at bottom layer to even wear over the entire disk.

A fun story

- For my PhD built Hector/Hurricane.
- Based on VFS, built OO layer, and a NFS, but people wanted very high speed disks…
- Developed I/O board and wrote HFS
- System crashed every day or so (it was the HW), so wrote a fsck for HFS, all was fine… until the file system grew and grew… took hours to run…
- The log structured FS came out, and eureka

My solution

- All data/meta data written to new locations on disk
- Two superblocks, alternate between them.
- Superblock written after all intermediate blocks/metadata
- System just restored to the last superblock; always a consistent (perhaps old) version of the FS
- Recovery time went from 2 hours to 2 seconds

Other stuff

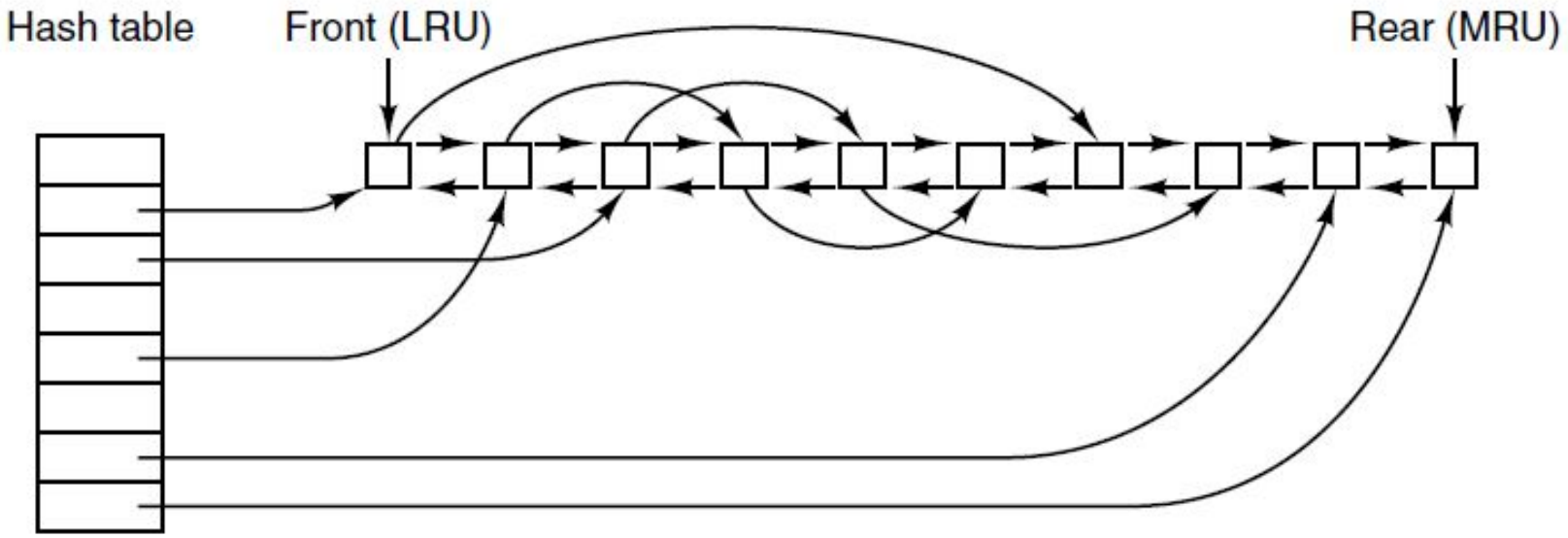
Backup: the book talks about…

- There are two reasons for backup:
  - Disaster recover
  - Dealing with Human stupidity
- Book spends several pages, describing important OS concepts:
  - logical versus physical
  - incremental versus full
  - offsite/security

NO!!! Computers cattle not pets

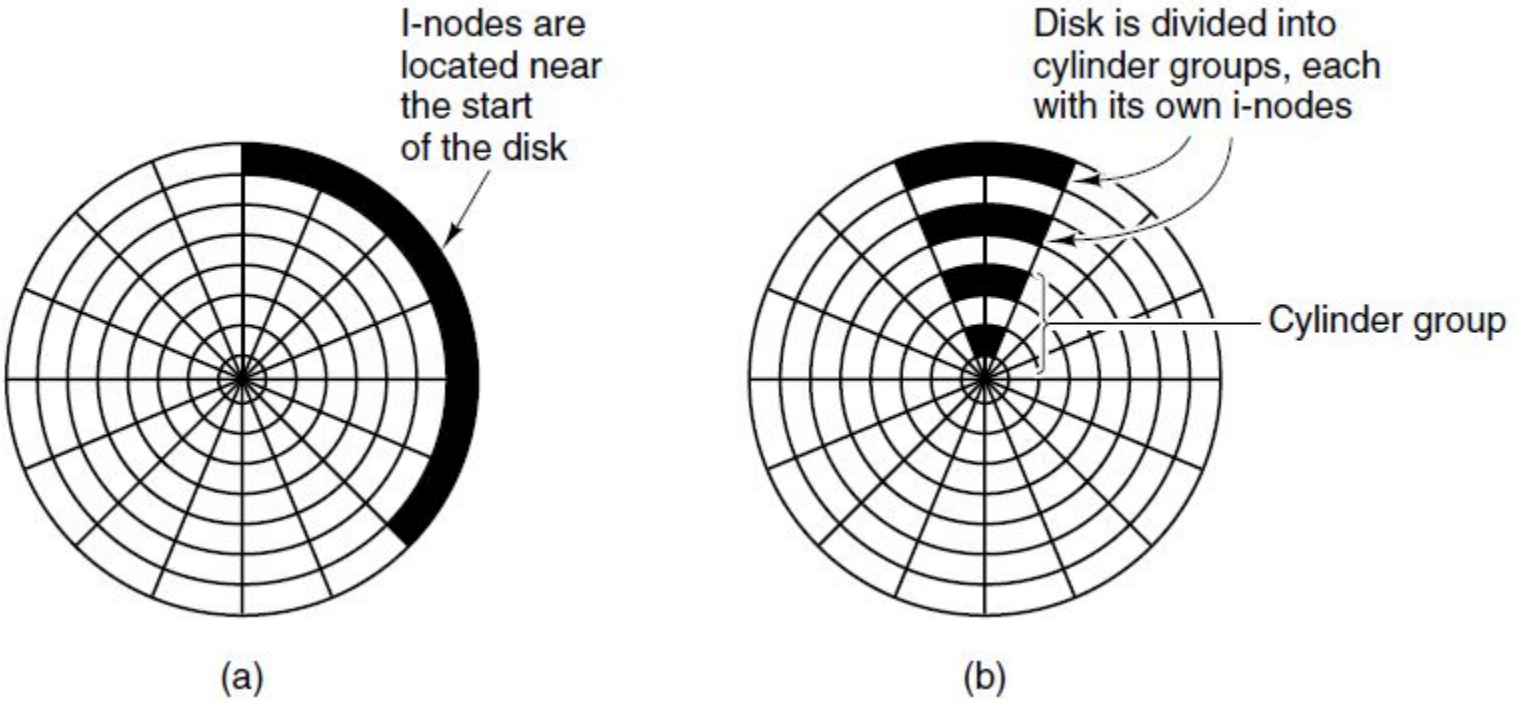
- data stored in DB with DB specific policies
- passwords in cloud password manager
- code in github
- music in cloud
- configuration is code...

Page cache



- We discussed when discussing MM, all modern OSes have a unified cache anonymous/file
- key point is that read/write references allow us to keep LRU
- Book sayed write out meta-data rapidly important… see log/soft update discussion

Thinking about the disk



- Putting inodes in fixed place great if you want to snap up a bunch, and if cache hit likely (a)
- Some file systems put in cylinder near blocks (or segments)