Operating Systems II

File Systems

File Systems: Motivation

Why do we need another sort of memory

Persistence ?

Sharing ?

Protection ?

Size ?

File Systems: System-oriented view

Files as general abstraction for long-lived system entities:

- ➡ user documents: regular files
- ➡ programs: executable file
- ➡ information to organize data: directories
- ➡ abstractions for storage devices: block files
- ➡ abstractions to model I/O-devices: special files

captured in the <u>file type</u>

File Systems: User-oriented view

User-oriented way of organizing data:

- user readable names
- different file types reflecting the kind of data
- functions to organize data
- functions to search data

desktop paradigm:

document, folder, waste basket, etc.

What is the difference to a database system?

Why is a file system part of an OS and a DBS not?

Naming

Examples

name.extension	Meaning
name.txt	Text file
name.c	C source file
name.o	Object file (machine code ut not linked)
name.bak	backup file
name. jpg	file coded in the JPEG standard
name.mp3	file coded in the MPEG 3 standard
name.pdf	pdf file (portbale document format)

gif, tiff, as, ps, zip, tex, hlp, html, doc, exe, xls......

Issues: Characters: upper/lower case, unicode,. .
Extensions: conventions vs. interpreted by the OS

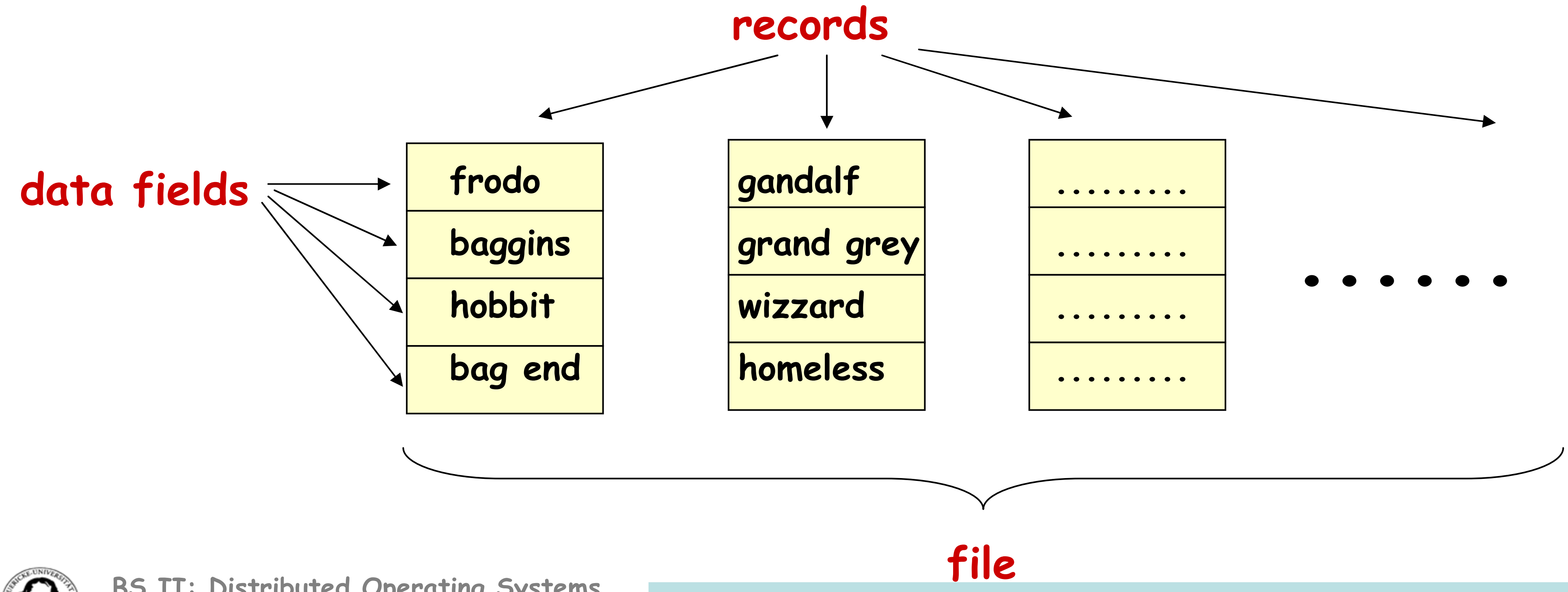
information structure

field: basic data element

record: set of related fields

file: set of similar records

example information structure: <first name>, <family name>, <origin>, <home address>



records

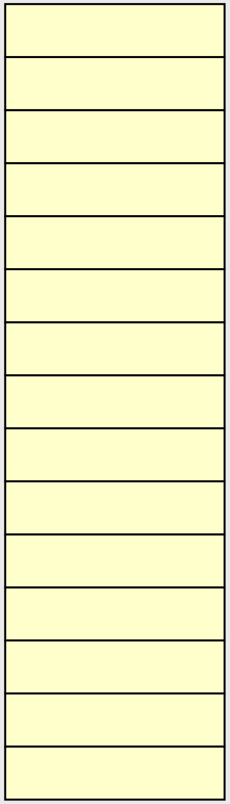
data fields

frodo	gandalf	....	......
baggins	grand grey	....	
hobbit	wizzard	....	
bag end	homeless	....	

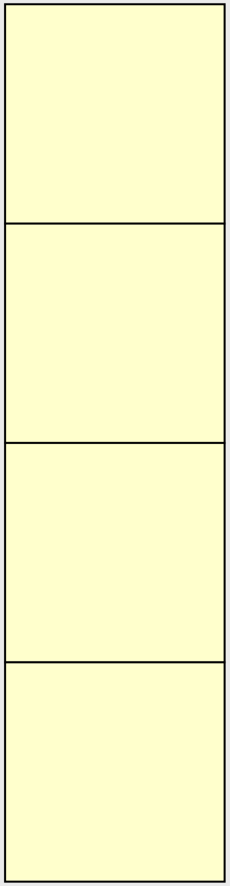
file

file organization

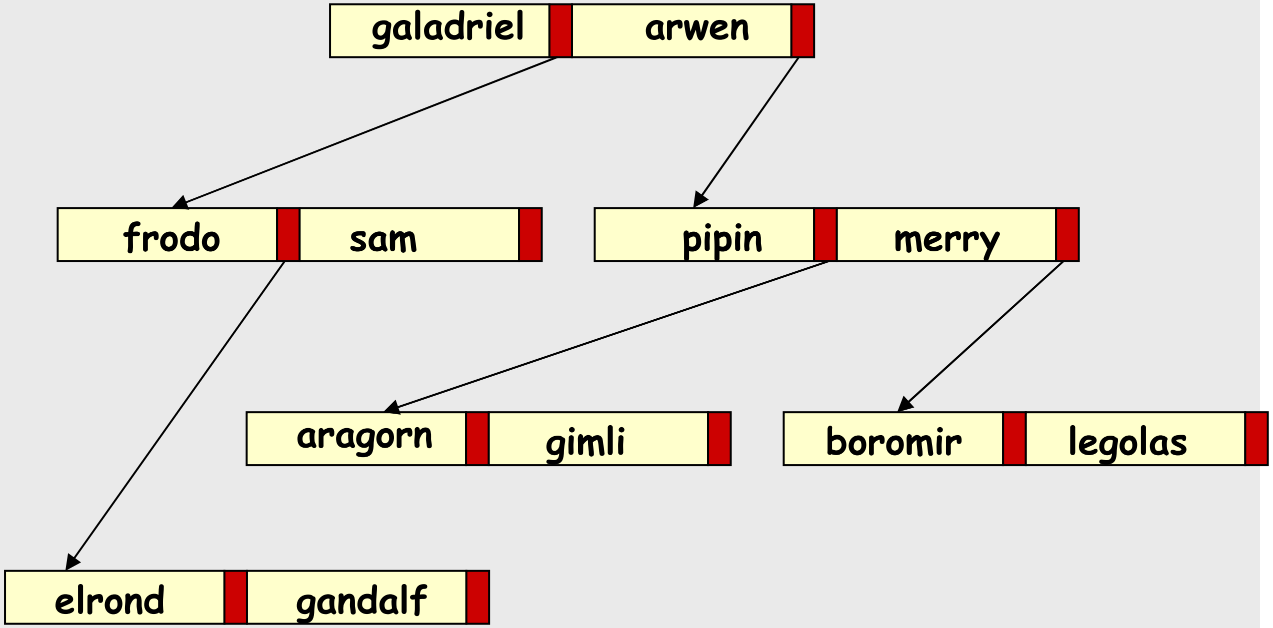
sequence
of bytes



sequence
of records



linked (tree)
structure



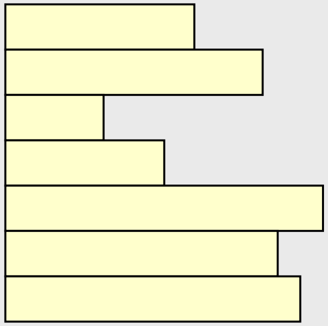
file organization and access

How to find a record? Alternatives in file organization:

collection:

variable length records

chronological order



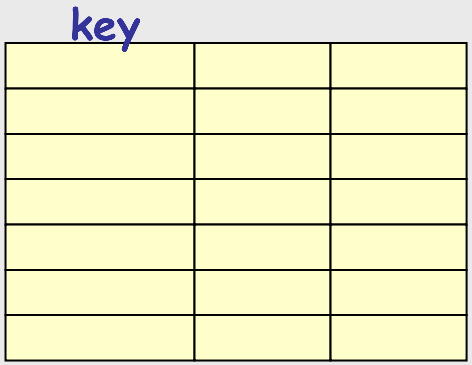
sequential:

fixed length records

sequential order

according to the

key field

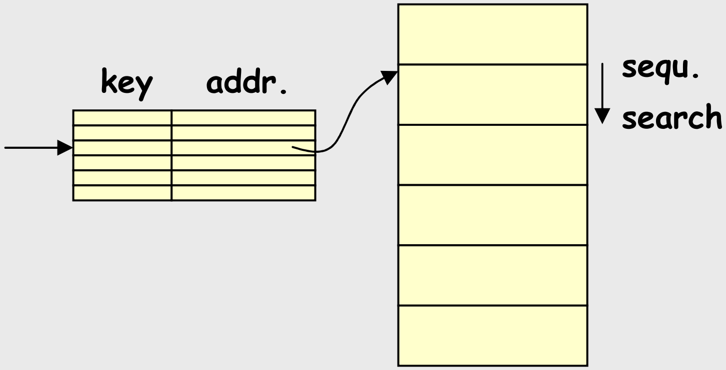


index sequential:

sequ. search in

the respective

section



fully indexed:

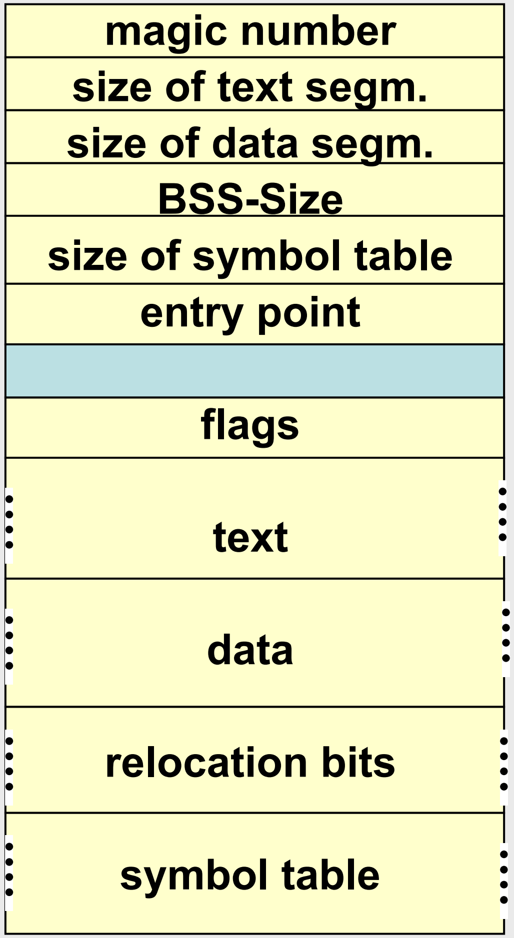
Index for multiple fields.

Index for each record.

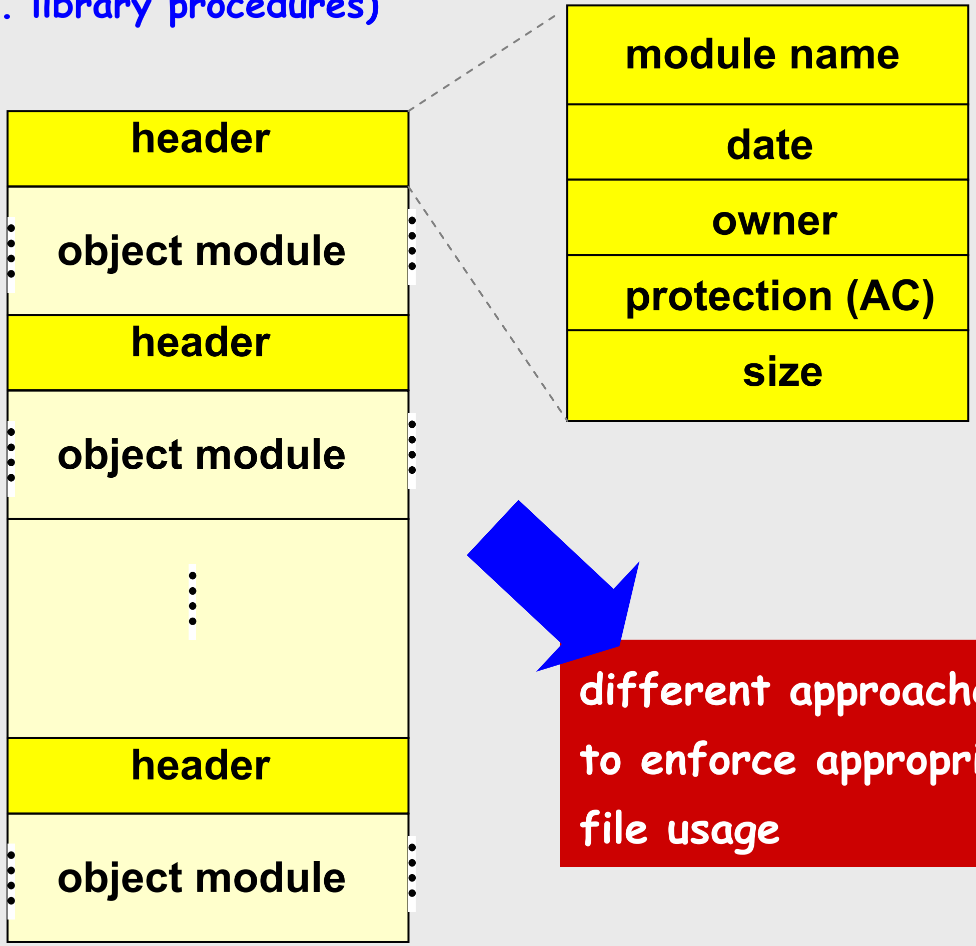
No sequential search.

file type

an executable file in UNIX



an archive of object modules
(e.g. library procedures)



different approaches to enforce appropriate file usage

file attributes

basic info: file name, file type, (file organization)

address info: device, phys. start address, act. size, max size

access control info: owner, access authorization, access rights

file info: creation data, creator ID, last read access,
ID of last reader, date of last modification,
ID of last writer, data of backup, actual use info.

file attributes

examples of file attributes

acces control	who is allowed to do what with the file
passwd	passwd for the file access
creator	ID of the file creator
owner	current owner
read-only-flag	0: R/W, 1:read only
hidden flag	0: default, 1: invisible
system flag	0: normal file, 1: system file
archive flag	0: changes saved, 1: not yet saved
ASCII/binary flag	0: ASCII file, 1: binary file
random access flag	0: sequential file, 1: random access
temporary flag	0: normal, 1: delete file on process termination
lock flags	0: not locked, ≠ 0: file locked
record length	number of bytes in a record
key position	offset to the key in the record
key length	number of bytes in the key
time of last access	date and time of last access to this file
time of last modification	data and time of last change
actual (max) size	number of (max) bytes in file

accessing a file: operations

explicit preparation for file acccess	create — delete open — close
normal file acccess	read — write append
search & management	seek set attributes — get attributes rename

```
/* program to copy file abc to file xyz. Error handling and report are minimal. */

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

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

#define BUF_SIZE 4096                                /* buffer of 4096 Byte */
#define OUTPUT_MODE 0700                             /* access rights of the new 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);                             /* if not exactly 3 arguments: syntax error */

/*open input file and create output file */
in_fd = open (argv[1], O_RDONLY);                   /* open source file (abc) */
if (in_fd < 0) exit (2);                             /* if not possible: quit */
out_fd = creat (argv[2], OUTPUT_MODE);              /* create target file */
if (out_fd < 0) exit (3);                           /* if not possible:quit */

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

Copy file abc to file xyz.
(Tanenbaum)

continue →

Copy file abc to file xyz. (Tanenbaum)

continued

```
/* close file */
close (in_fd);
close (out_fd);
if (rd_count == 0)
/* no error at last read */
                  exit (0);
else
                  exit (5);
/* error at last read */

}
```

Memory mapped files

Idea: Map files to virtual memory. Exploit paging mechanism to swap data between disk and physical memory.

Benefit: file can be accessed by normal (memory) read and write operations.

System calls:

map (virtual address): maps file to virtual address space starting at virtual address.

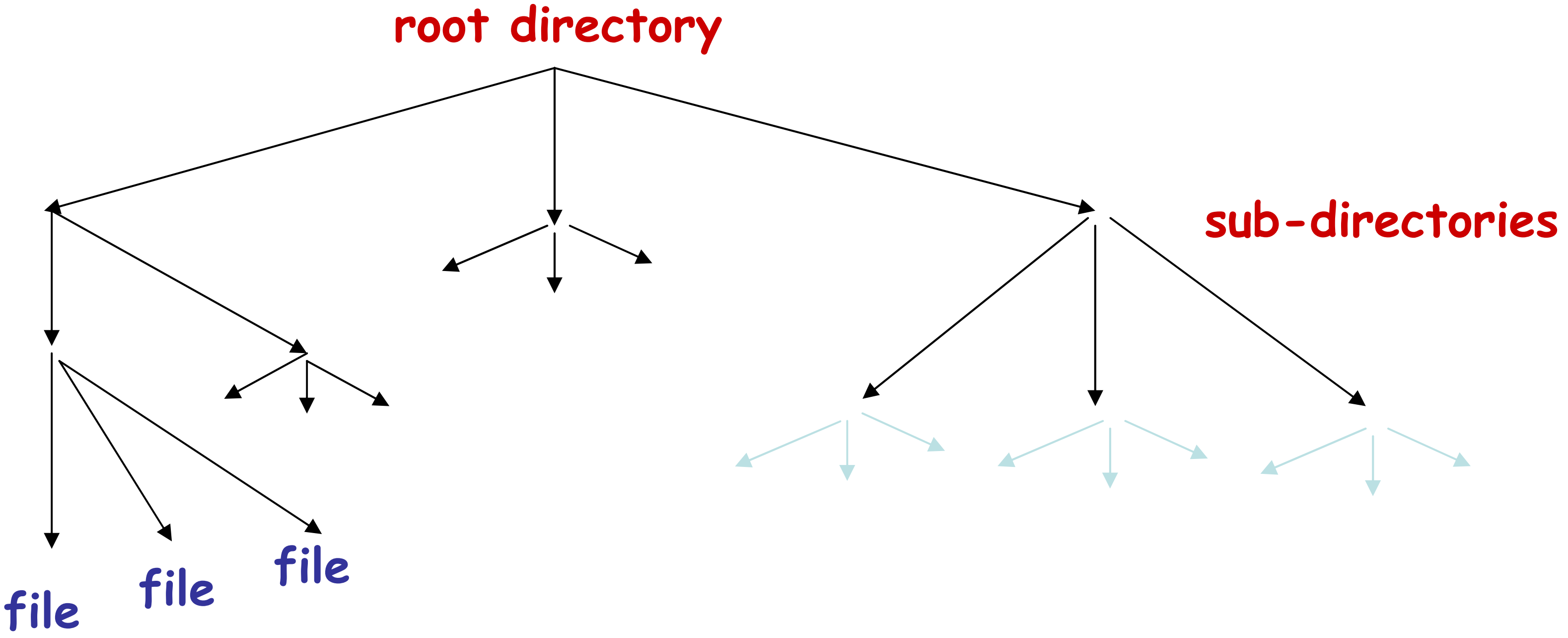
unmap: remove file from virtual address space.

Problems: exact size of output file, sharing of memory mapped files, file may not fit into virtual address space.

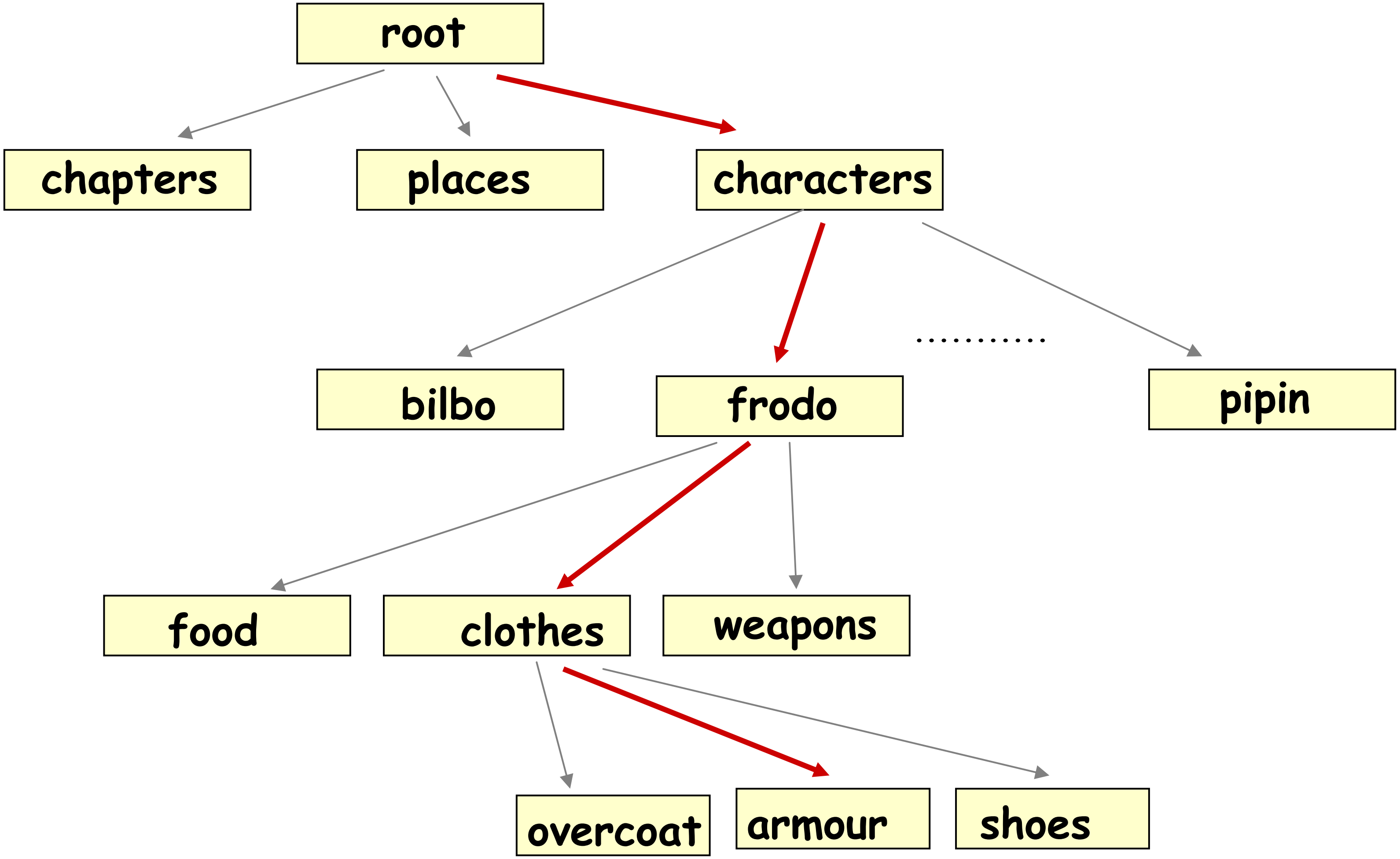
organizing files: directories or folders

Single level directory: organized along users, simple and easy to implement

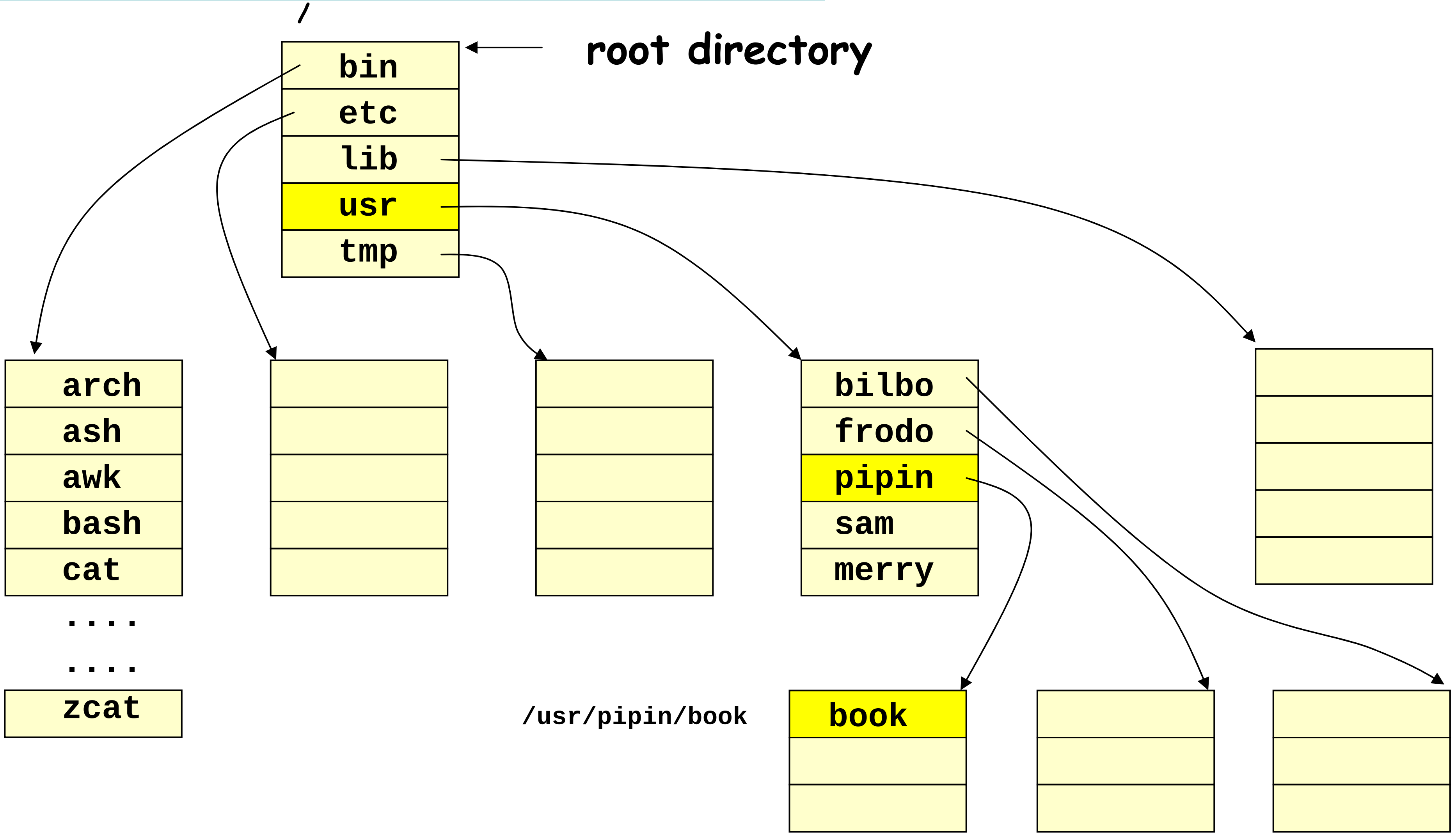
Hierarchical directory structure:



hierarchical directories and pathnames



UNIX directory tree



hierarchical directories and pathnames

example dialogue in Unix: typed commands, response

```
cd /
pwd
/
ls
bin  boot  dev  etc  home  lib  lost+found  tmp  usr  var
cd
pwd
/usr/kaiser
ls -all
drwxr-xr-x  14 kaiser root    4096 March 22 18:17   .
drwxr-xr-x   3 root   root    4096 Dec   11 2003    ..
-rw-------   1 kaiser usr   742068 Nov   13 2004    pubsub-12112003.tar.gz
....
....
cd ..
pwd
/usr
```

operations on directories

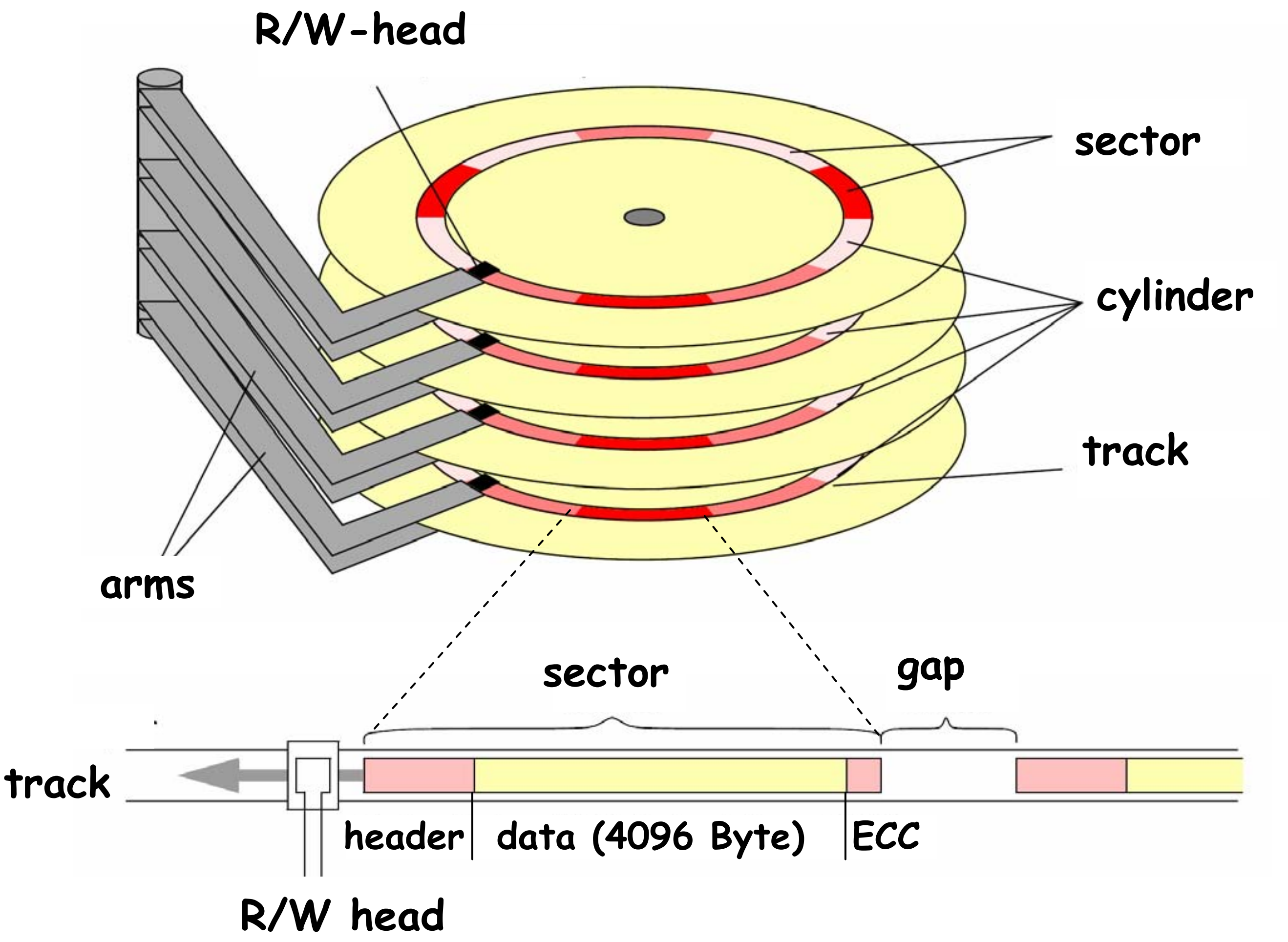
- creat(e)
- delete
- opendir
- closedir
- readdir
- rename
- link
- unlink

File system implementation

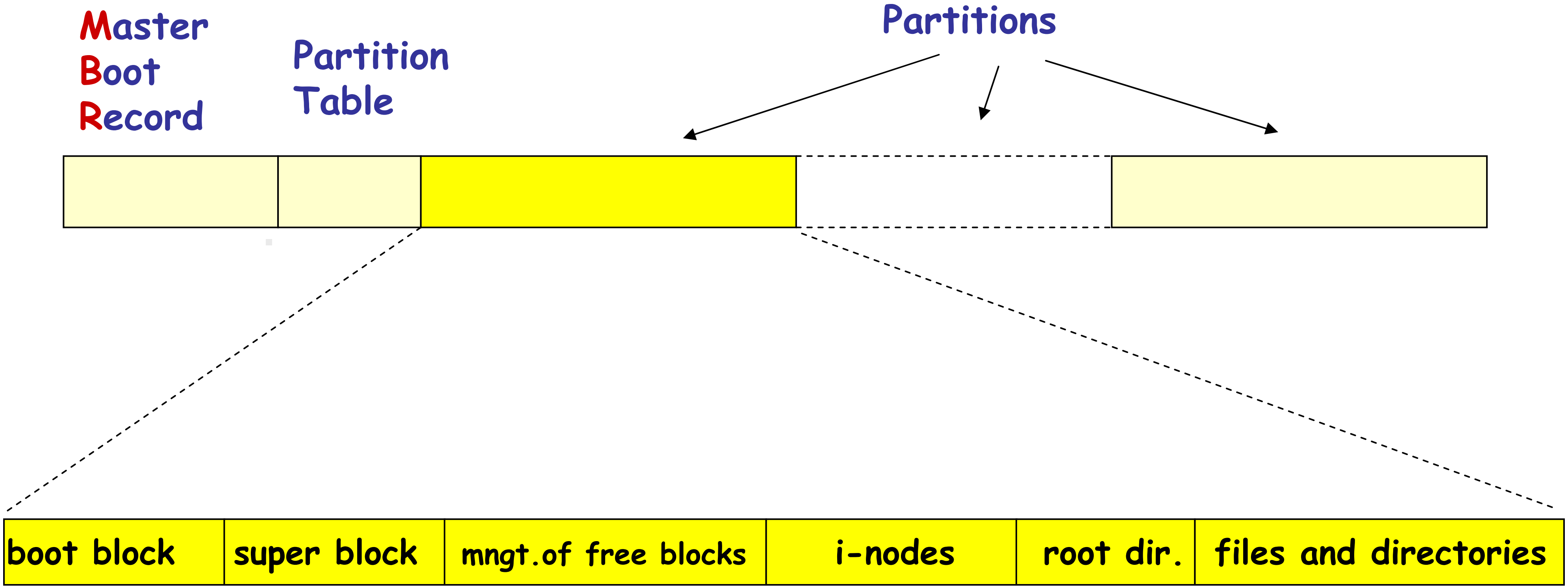
Issues:

- how to map files to disk blocks
- how to find the respective disk blocks
- how to realize directories
- how to share files

recall: the physical organization of a disk



file system layout



Master Boot Record

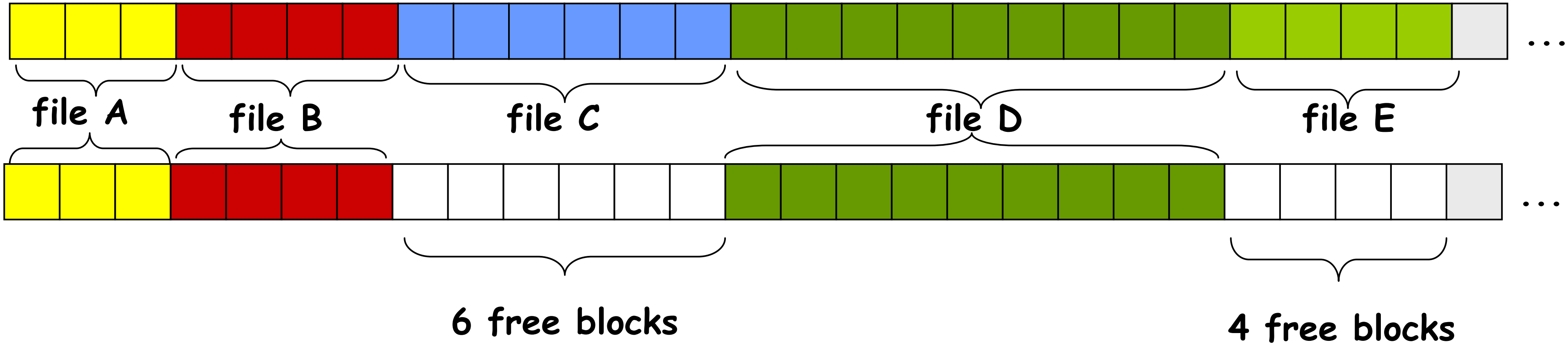
Partition Table

Partitions

boot block	super block	mngt.of free blocks	i-nodes	root dir.	files and directories

Variations in organizing the files system

continuous allocation of disk blocks



pro: simple implementation

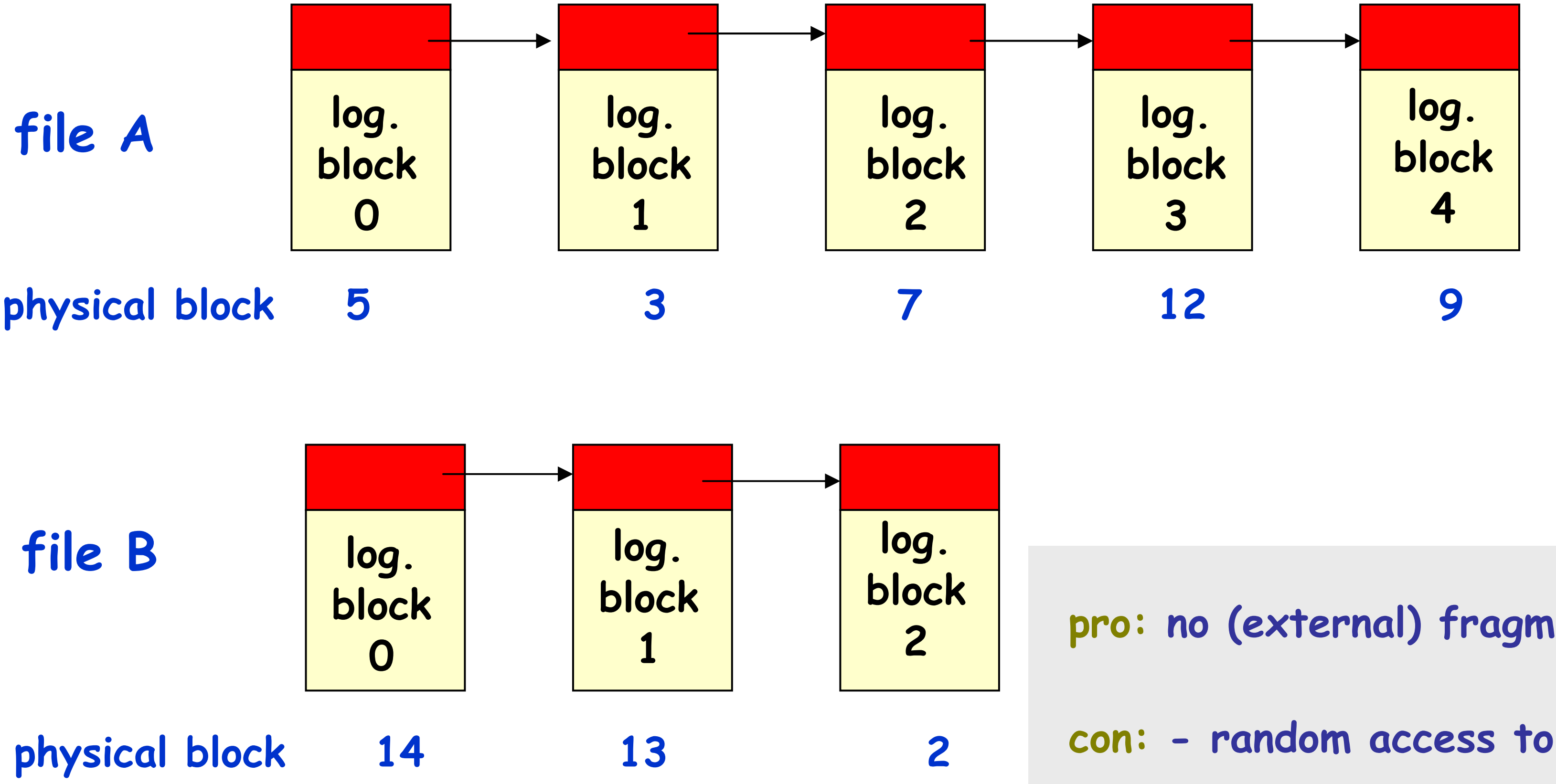
good read performance

con: file size must be known in advance

identification and reuse of free parts

Variations in organizing the files system

linked list



pro: no (external) fragmentation

con:
- random access to blocks is slow
- usable bytes in block ≠ 2^n

Variations in organizing the files system

linked list implemented by a File Allocation Table (FAT) in memory

physical block no.

0	
1	
2	-1
3	7
4	
5	3
6	
7	11
8	
9	-1
10	
11	9
12	
13	2
14	13
15	

file A starts in physical block 5

file B starts in physical block 14

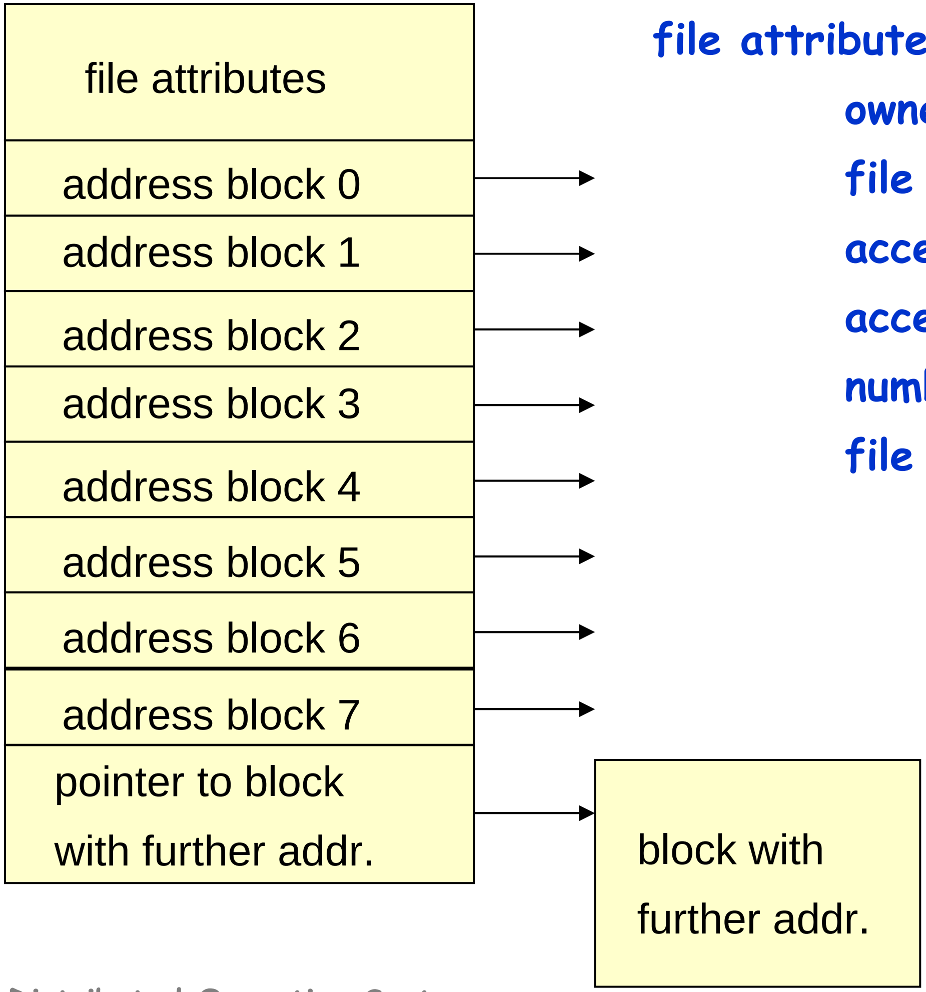
-1: file termination symbol

pro: omits random access problems of linked list

con: size is proportional to disk size.

Variations in organizing the files system

i-node, inode, index node



file attributes

address block 0

address block 1

address block 2

address block 3

address block 4

address block 5

address block 6

address block 7

pointer to block with further addr.

block with further addr.

file attributes are e.g.:

- owner
- file type
- access permissions
- access time
- number of links to the file
- file size

pro:
- Only the inodes of open files need main memory. No relation to disk size.
- Allows to store attributes and file data separately.

implementing directories

what information is needed in a directory entry?

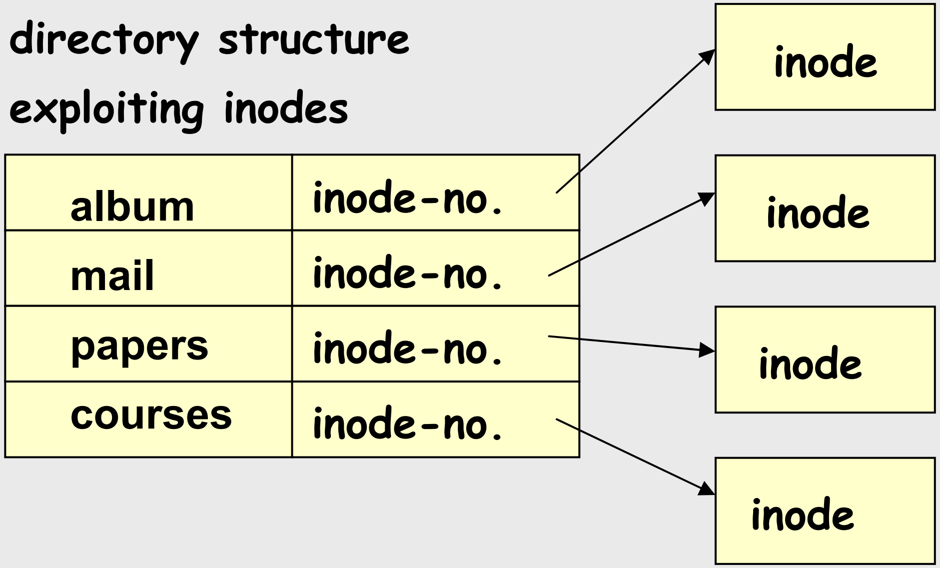
- information about the file type
- how to find a file on the disk
- additional information

} file attributes

simple directory with fixed length entries

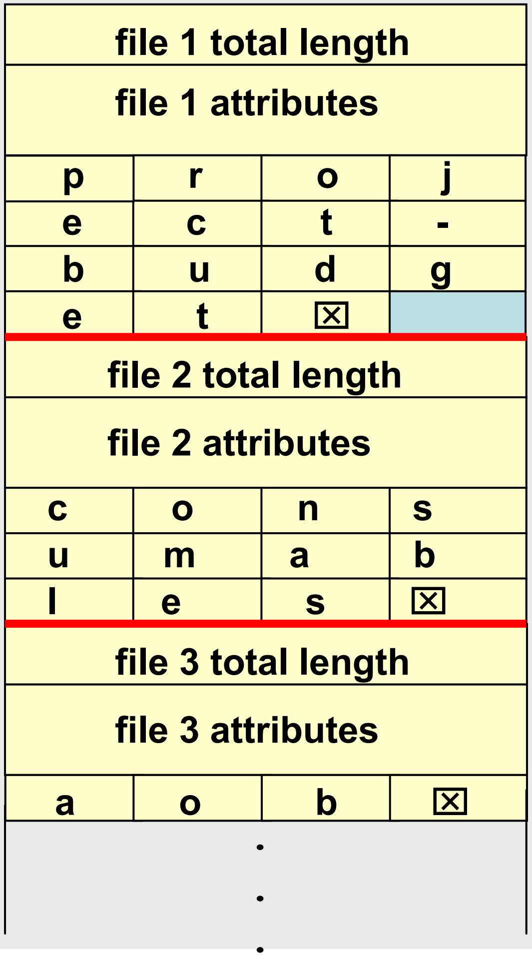
album	attributes
mail	attributes
papers	attributes
courses	attributes

directory structure exploiting inodes



album	inode-no.
mail	inode-no.
papers	inode-no.
courses	inode-no.

handling long file names



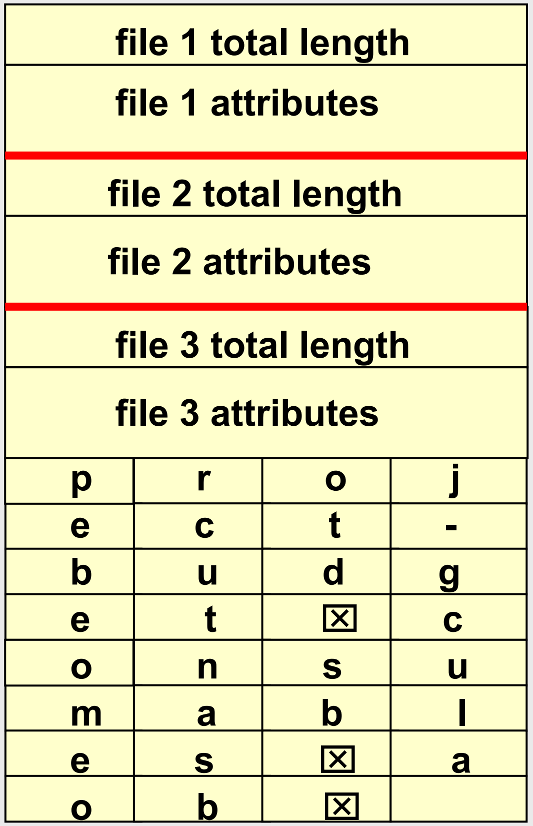
in-line

file 1 total length			
file 1 attributes			
p	r	o	j
e	c	t	-
b	u	d	g
e	t	☒	
file 2 total length			
file 2 attributes			
c	o	n	s
u	m	a	b
l	e	s	☒
file 3 total length			
file 3 attributes			
a	o	b	☒

.

.

.



on a heap

file 1 total length			
file 1 attributes			
file 2 total length			
file 2 attributes			
file 3 total length			
file 3 attributes			
p	r	o	j
e	c	t	-
b	u	d	g
e	t	☒	c
o	n	s	u
m	a	b	l
e	s	☒	a
o	b	☒	

☒: termination symbol

file names may be variable length 1-255 characters.

problem:

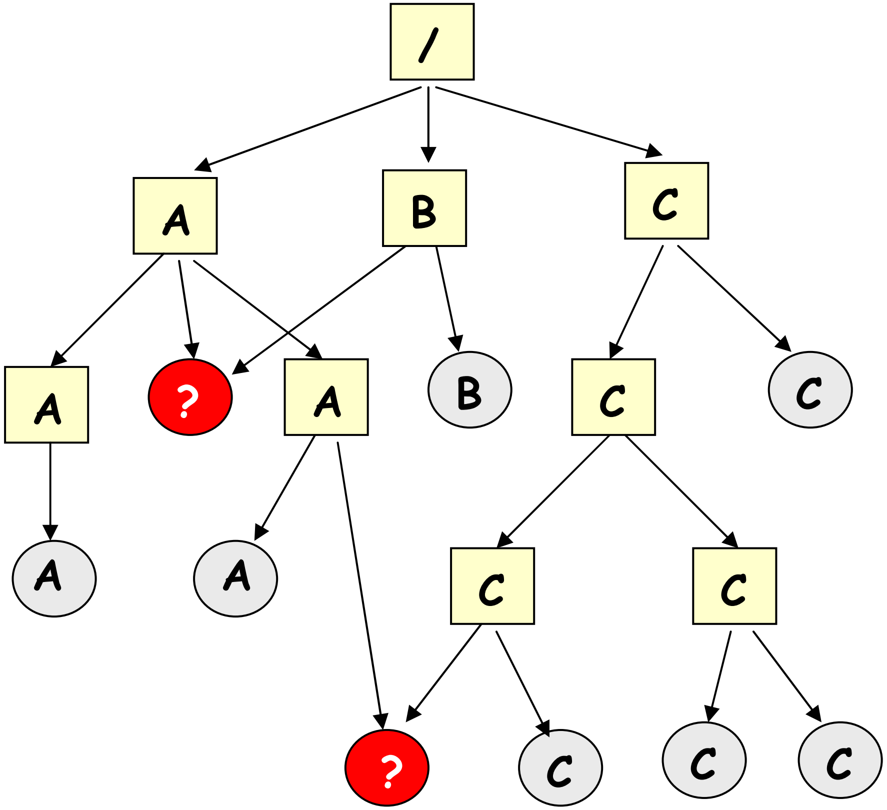
linear serach

improvements:

hashing

caching

sharing files



Directed Acyclic Graph

🟨 directory

○ file

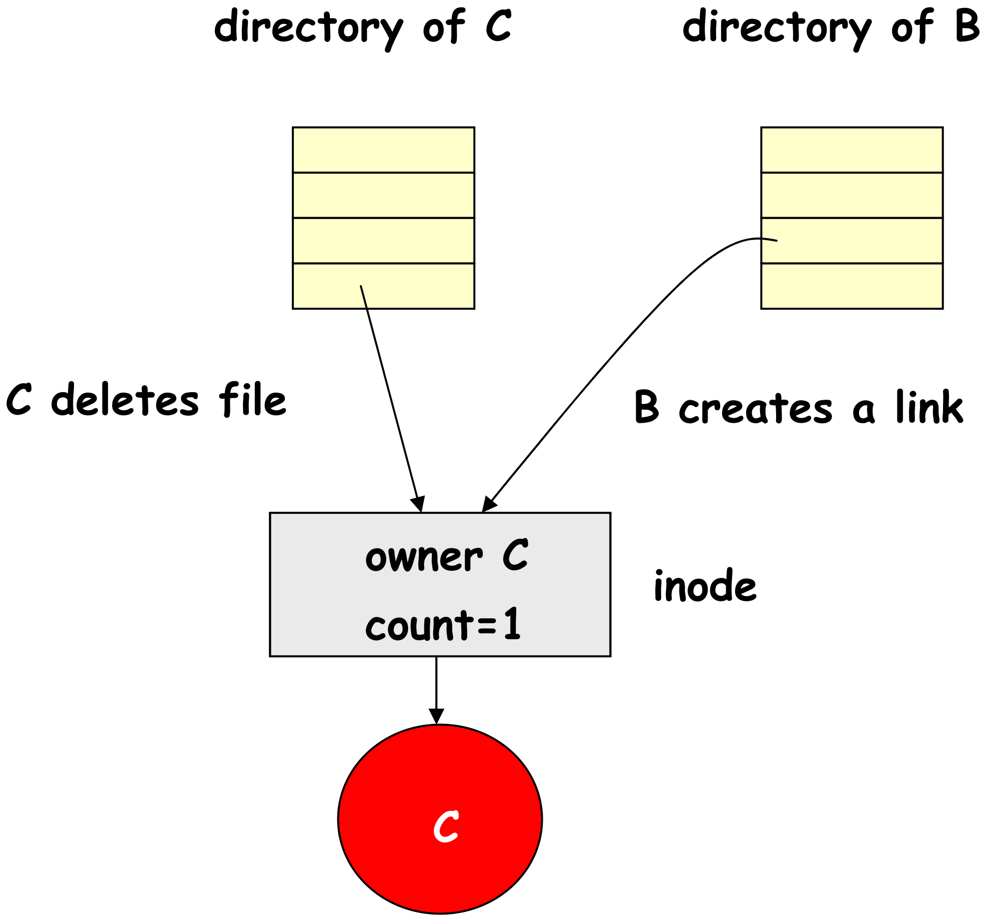
🟥 shared file

A,B,C : owner

problems:

- who is the owner of a shared file?
- how to ensure visibility of changes?

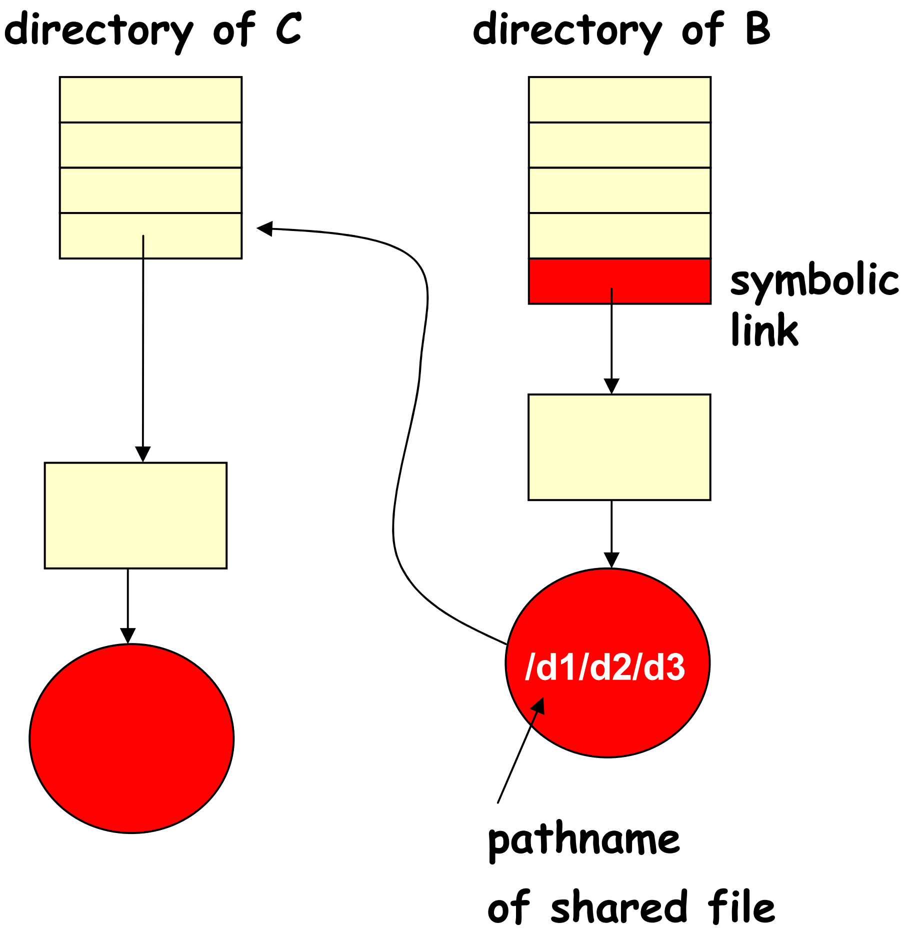
sharing files



problem:

B is the only user of a file of owner C.

sharing files by symbolic links



owner has full control over file

problem:overhead

- analyzing and following the path requires additional disk accesses.

- additional inode for every symbolic link.

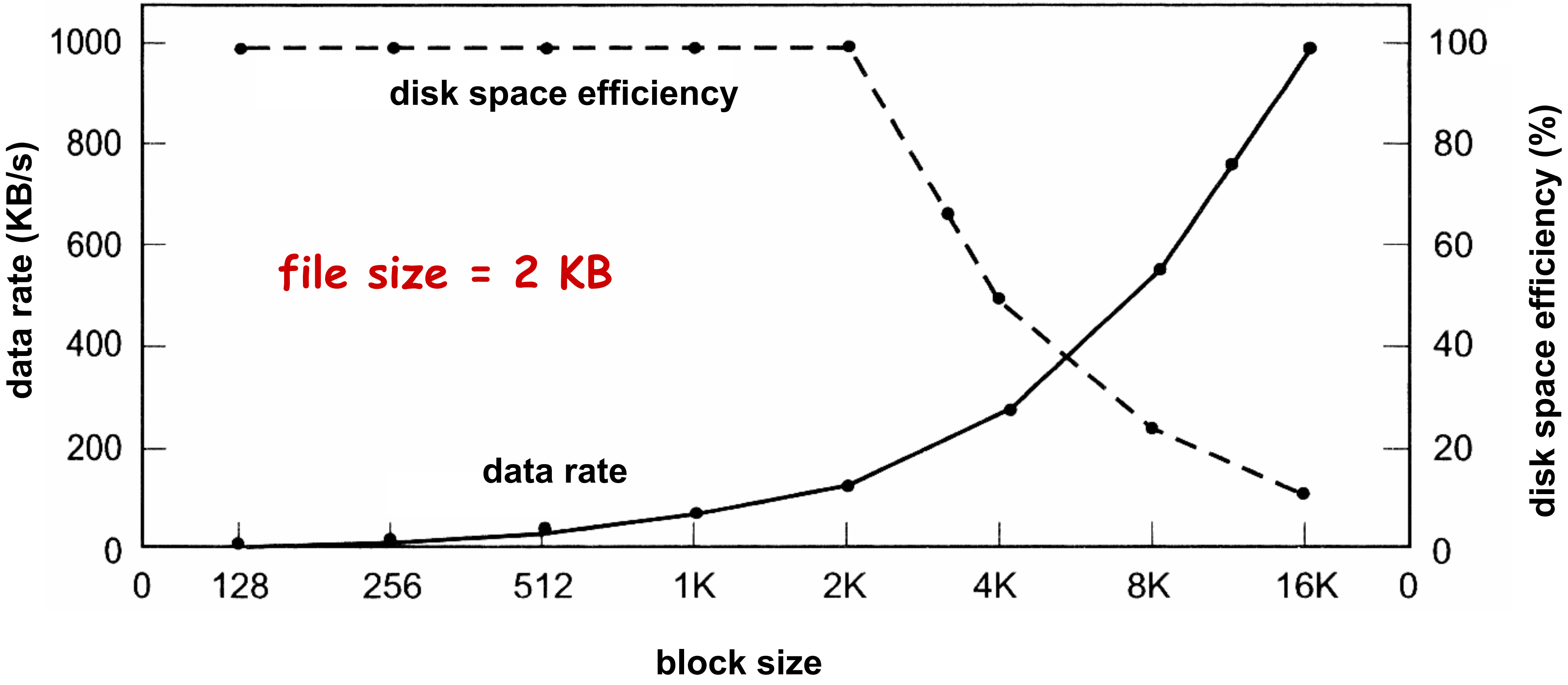
managing the disk

issues:

- managing the physical disk space
    - block size
    - allocating free blocks
    - disk quotas
- reliability
    - backups and recovery
    - consistency
- file system performance
    - caching
    - block ahead read

managing the disk

impact of block size on space efficiency and data rate



(Tanenbaum 2003)

managing the disk

1. Linked list of free blocks

size of list and max. space requirements:

16 GB disk, block size 1k:

--> 16M entries by 32 bit

--> 1 block 255 (+1 to link the blocks) entries --> ~ 40 K blocks

changes over time when
more disk space is allocated

2. bit map of free blocks

size of list and max. space requirements:

16 GB disk, block size 1k:

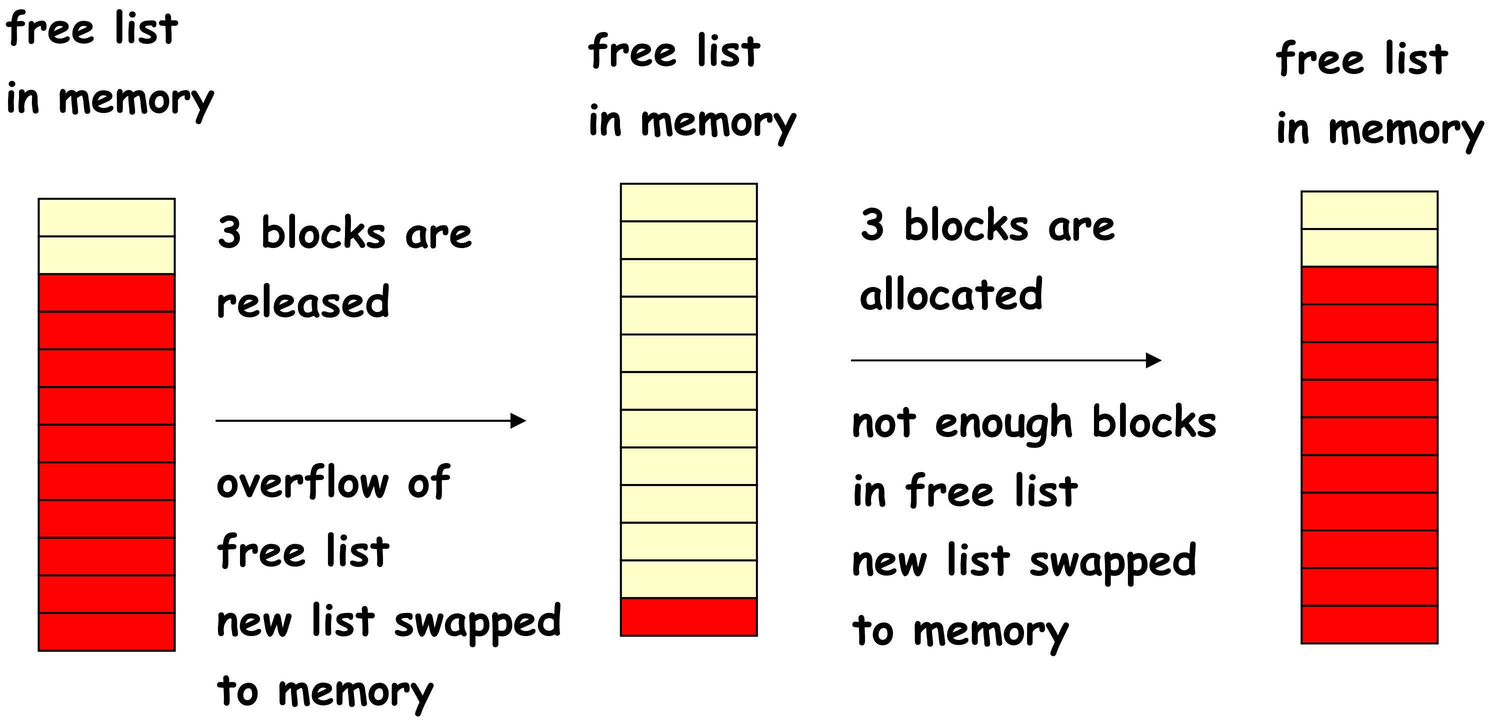
--> 16M entries by 1 bit

--> 1 block 1k x 8 bits --> ~ 2K blocks

fixed over time

managing the disk

problem with caching of free entries in main memory



free list
in memory

3 blocks are
released

overflow of
free list
new list swapped
to memory

free list
in memory

3 blocks are
allocated

not enough blocks
in free list
new list swapped
to memory

free list
in memory

pointer to free blocks 🟥

managing the disk

disk quotas restrict disk space on a per user base.

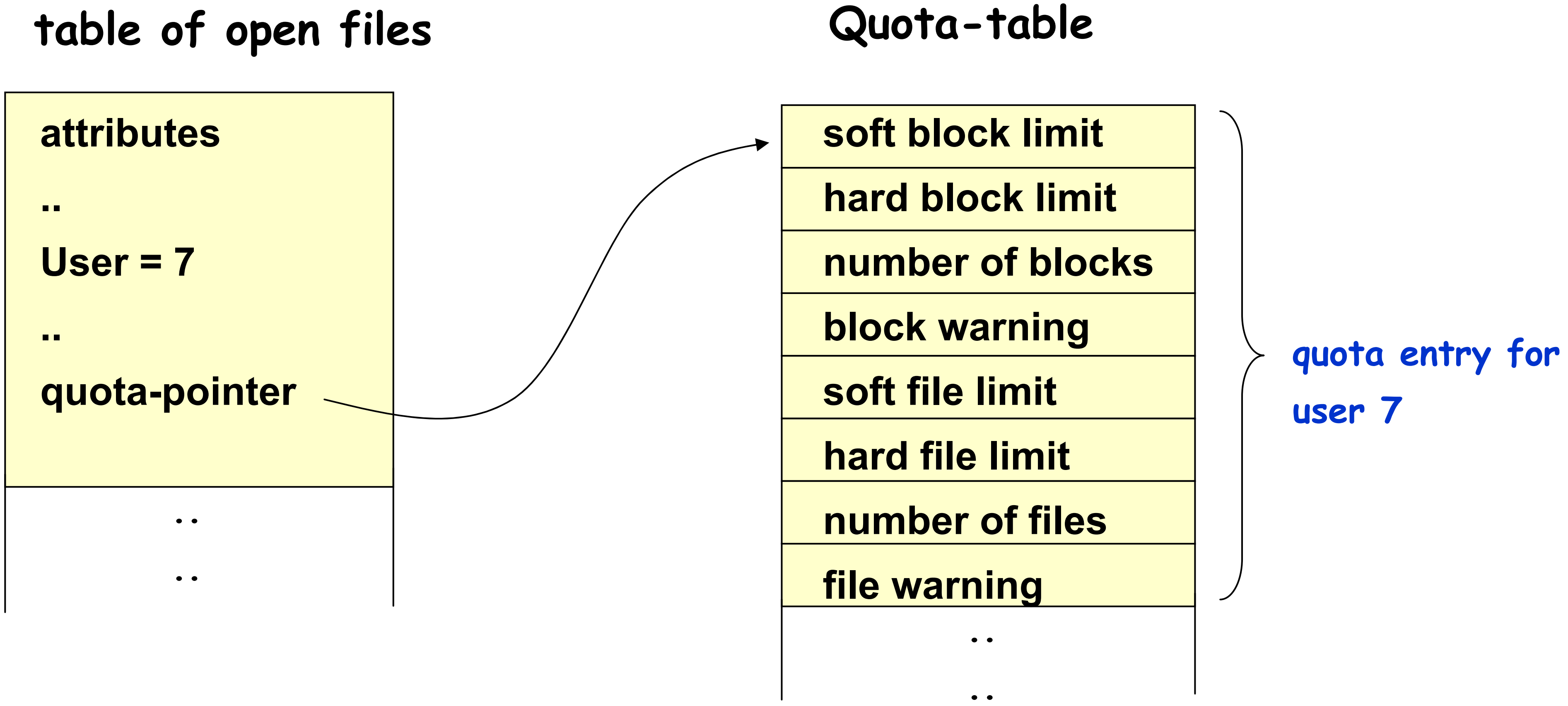


table of open files

attributes
..
User = 7
..
quota-pointer

..
..

Quota-table

soft block limit
hard block limit
number of blocks
block warning
soft file limit
hard file limit
number of files
file warning

..
..

quota entry for user 7

Robustness and Dependability of a File System

Loss of Data is the "Super GAU" in a computer system!

While the cost of a new computer is in the order of 5.000 €
the cost of lost date may easily be higher many orders of magnitudes !

File system must be protected against:

- disk crashes
- erroneous software
- malicious accesses

Robustness and Dependability of a File System

Impairment	Countermeasures
defective blocks from manufacturing	directory of bad blocks on medium
transient reading and writing errors	code redundancy
physical destruction of disk	backup on redundant medium, mirrored disk (e.g. RAID 2), data replication,
software faults	user related access rights, least priviledge
system crashes	fsck, scandisk, journaled file systems
malicious accesses	access protection, encryption, fragmentation
erroneous deletion of files	no physical deletion, backups

Backup copies

physical backup: copies all blocks of the disk to the backup medium.

pro: simple

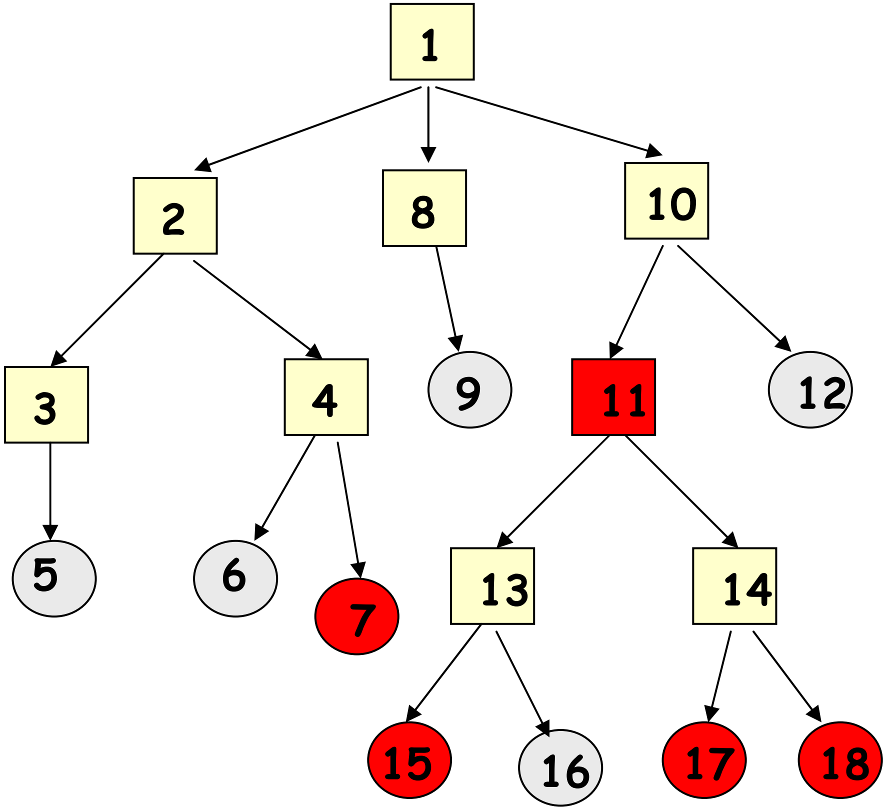
con: saves free blocks, problems with bad blocks, complete backup only.

logical backup: based on the file system structure. Recursively saves directories and files starting at user selected dir's.

pro: incremental algorithm only saves changes since last backup.

con: more complicated implementation.

incremental backup



🟨 unmodified directory

○ file: unmodified since last backup

🔴 🟥 modified since last backup

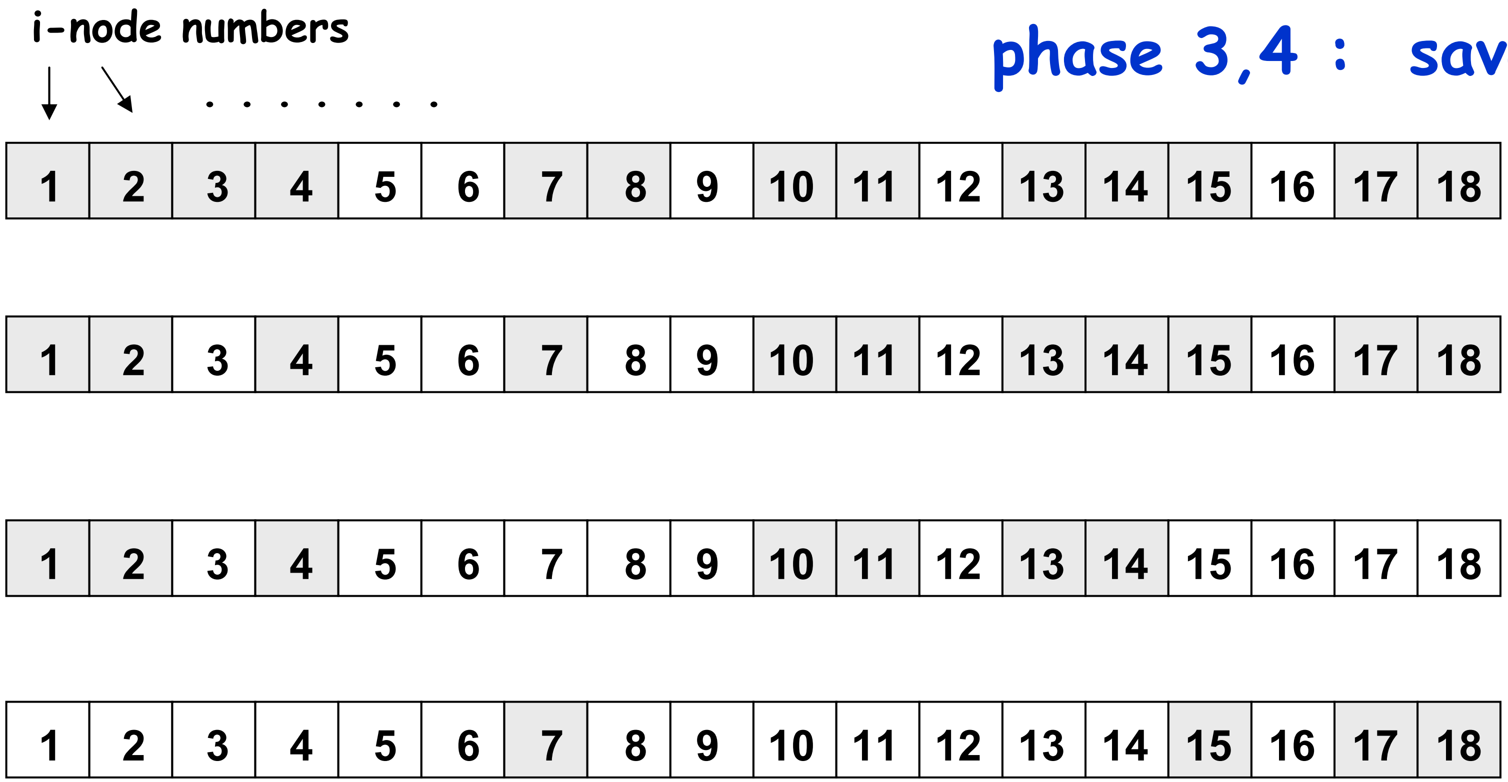
Incremental backup:

- exploits time and date to save modifications since last backup
- saves the entire path to the modified files including directories even when they didn'd change.

incremental backup

phase 1,2 : mark

phase 3,4 : save



1. mark modified files and all dir's
2. unmark dir's to unmodified files
3. store marked directories
4. store marked files

- the scheme stores all needed directories on the backup record first.
- during recovery they will occur first on the sequential medium and restored first.

file backup

Issues to be considered:

1. List of free blocks is a data structure in volatile memory and has to be rebuilt.
2. Multiple links to a file. This file has to be restored only once but the link has to be re-established in all directories.
3. Sparsely used files with holes.
4. Special files as pipes and device specific files should not be backed up.

file system consistency

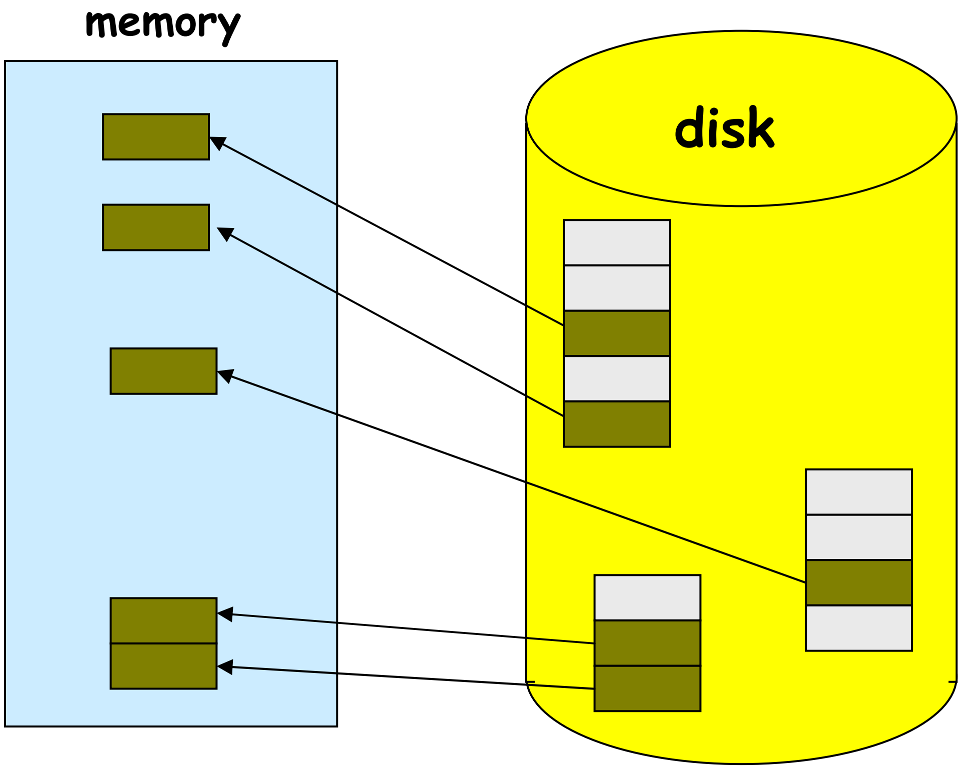
Changes on files are made in volatile fast memory and are not immediately stored on disk persistently.

file images
(some blocks of a file)

directory images
(some blocks of a directory)

i-node images
(some blocks of the inode table)

free list images
(some blocks of the free list)



file system consistency

after a crash...

First goal: maintain the consistency of the meta-data, i.e. all data structures which are involved in the management of the file system. E.g. i-nodes, directories, free-lists.

→ Exploit redundancy in the file system organization.

Normally not considered: modifications on file data. They are lost.

→ Journaled File Systems, Data Bases

file system consistency

fsck: file system check

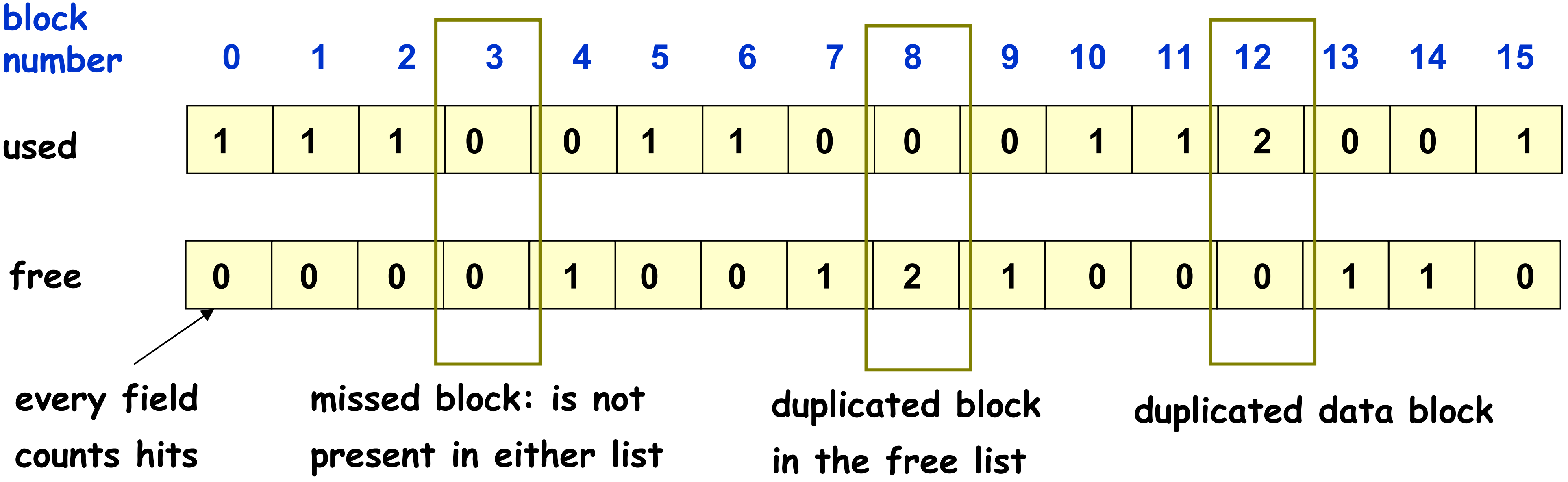
checks file system meta data on consistency.

1. missed or duplicated blocks
2. directory structure

file system consistency

Missed or duplicated blocks: fsck

1. scans all inodes to build the list of used blocks
2. scans the free list or bit map to find the free blocks



block number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
used	1	1	1	0	0	1	1	0	0	0	1	1	2	0	0	1
free	0	0	0	0	1	0	0	1	2	1	0	0	0	1	1	0

every field counts hits

missed block: is not present in either list

duplicated block in the free list

duplicated data block

file system consistency

Case 1: Missed Block

block number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
used	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	1
free	0	0	0	1	1	0	0	1	2	1	0	0	1	1	1	0

Problem: reduced disk capacity

Solution: Assign missed blocks to free list

file system consistency

Case 2: Duplicated block in the free list

block number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
used	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0	1
free	0	0	0	1	1	0	0	1	1	1	0	0	1	1	1	0

Solution: Rebuild free list and delete duplicated entry

file system consistency

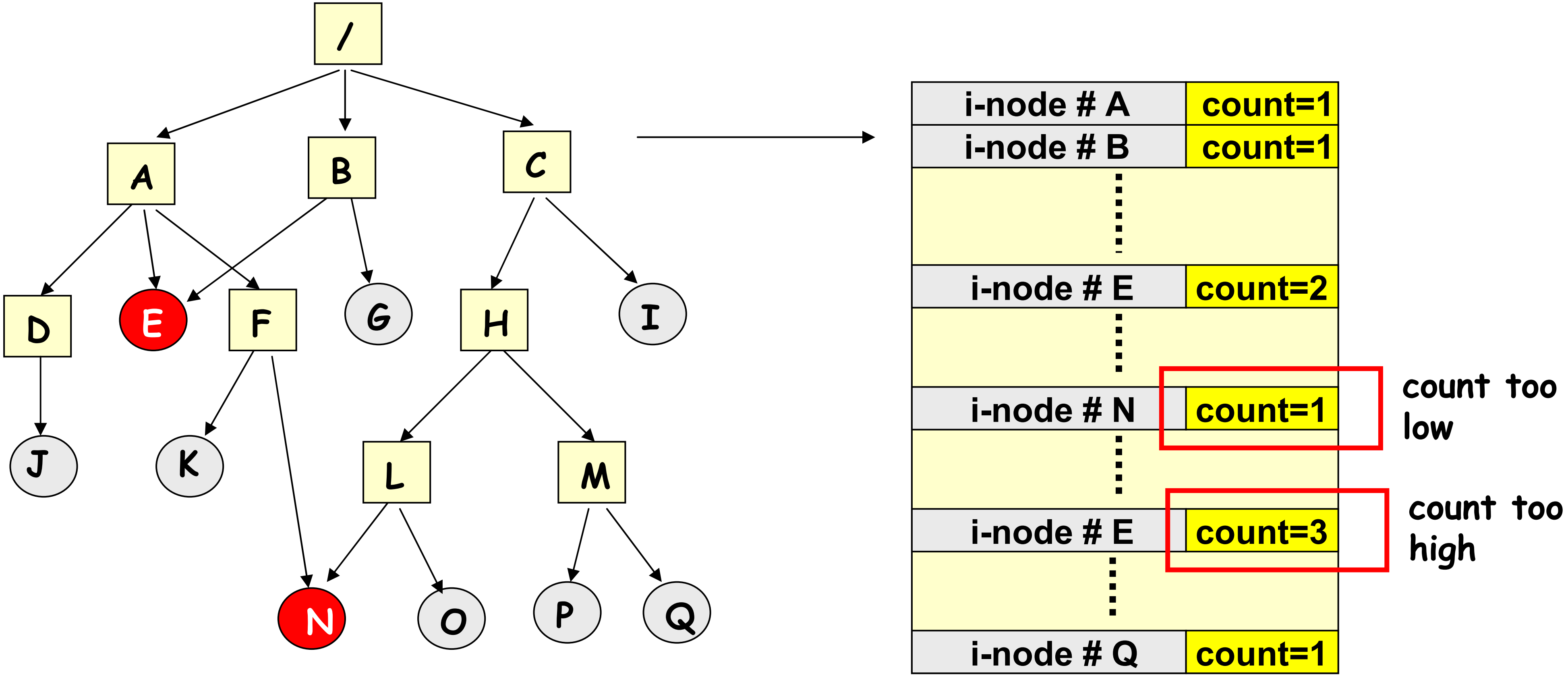
Case 3: Duplicated data block, i.e. block occurs in two files.

block number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
used	1	1	1	1	0	1	1	0	0	0	1	1	1	0	0	1
free	0	0	0	0	1	0	0	1	2	1	0	0	0	1	1	0

Problem: simple deletion results in further inconsistencies

Solution: copy one block to a free block and update the lists.

checking the directory system



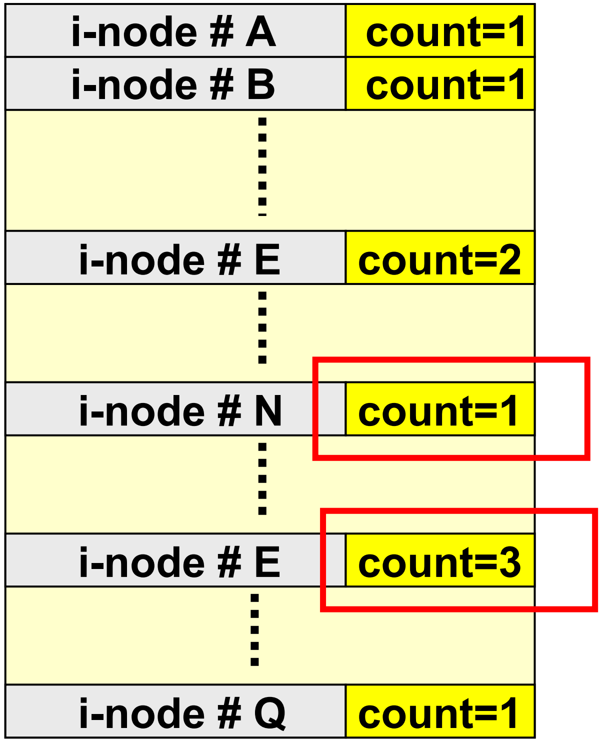
i-node # A	count=1
i-node # B	count=1
⋮	
i-node # E	count=2
⋮	
i-node # N	count=1
⋮	
i-node # E	count=3
⋮	
i-node # Q	count=1

count too low

count too high

1. step: build a list indexed by i-node numbers and count the occurence of every file in every directory.
2. step: compare the list count with the link counter in the i-node entries of files.

checking the directory system



link count in i-node
is higher than act. count in list

non-critical: i-node remains existent even when all links to a file in the directories are removed.
--> a space/efficiency problem

link count in i-node
is lower than act. count in list

critical: i-node will be deleted even if there exists a link to the file in some directory. When link counter goes to "0" the file system marks i-node as free and releases associated blocks.

improving file system performance

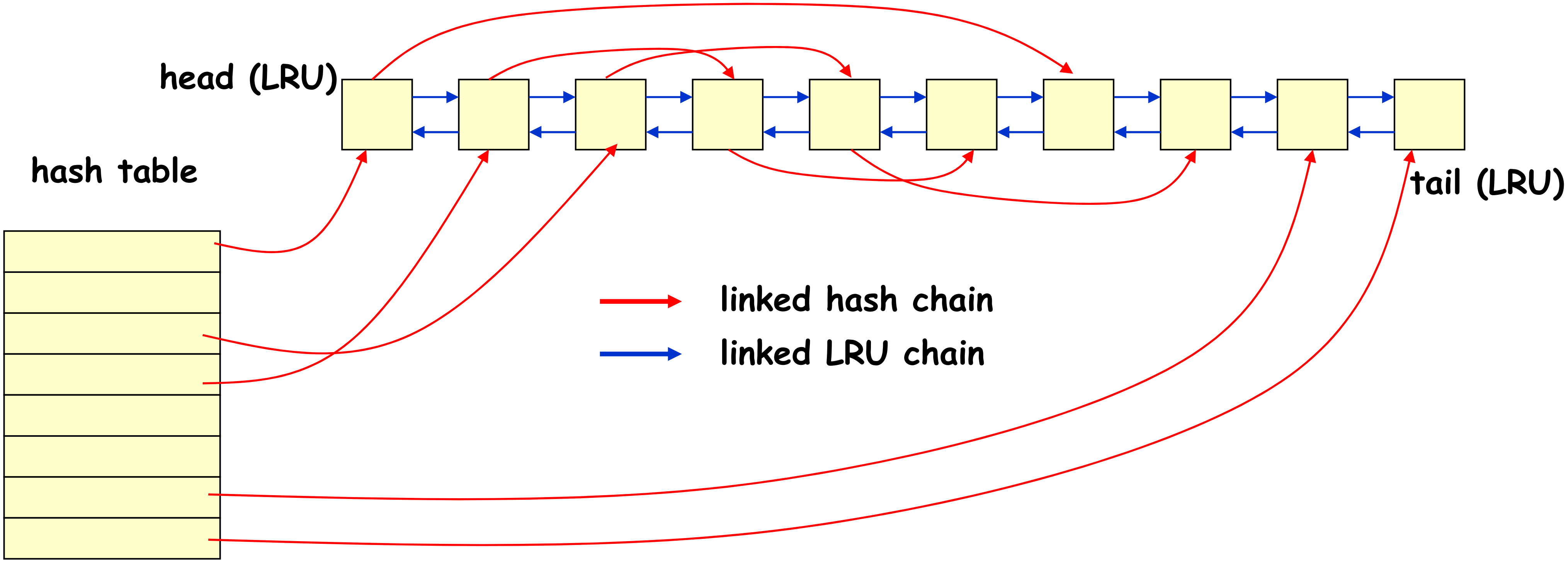
- caching
- block read ahead
- optimizing disk head movements
- log-based file systems

the buffer cache

Problem: access to main memory is up to 6 orders of magnitute faster than a disk access

- map files to virtual memory.
  - under explicit progr. control
- treat main memory as a cache for the disk.
  - transparent
  - similarities to virtual memory management.

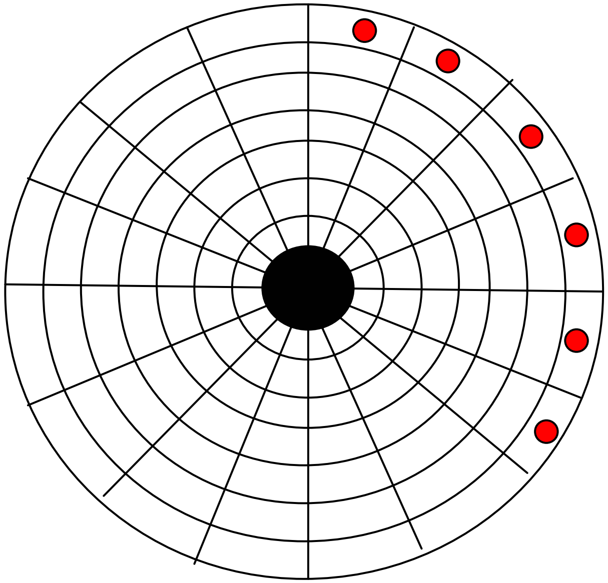
the buffer cache

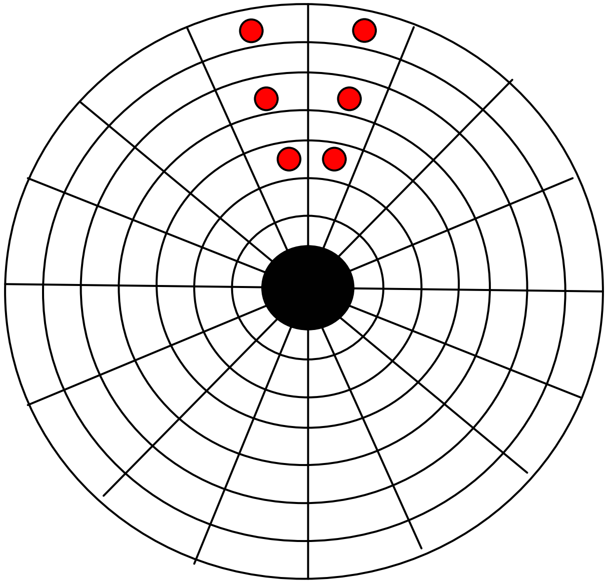


Problem: block contents in memory and block contents on disk are not identical.

- inconsistencies in case of crashes.
- trade-off between frequent disk updates and loss of data.
- explicit synchronization (sync).

optimizing disk access





i-nodes at the beginning of the disk.
distance between i-node and asociated blocks: number of cylinders/2

i-nodes and asociated blocks are organized in cylinder groups.

Log structured file systems

Motivation:

CPU performance
disk capacity
main memory capacity

} grow rapidly

Problem: disk access time doesn't improve much (seek ~10ms, wait ~4ms, write 50µs).

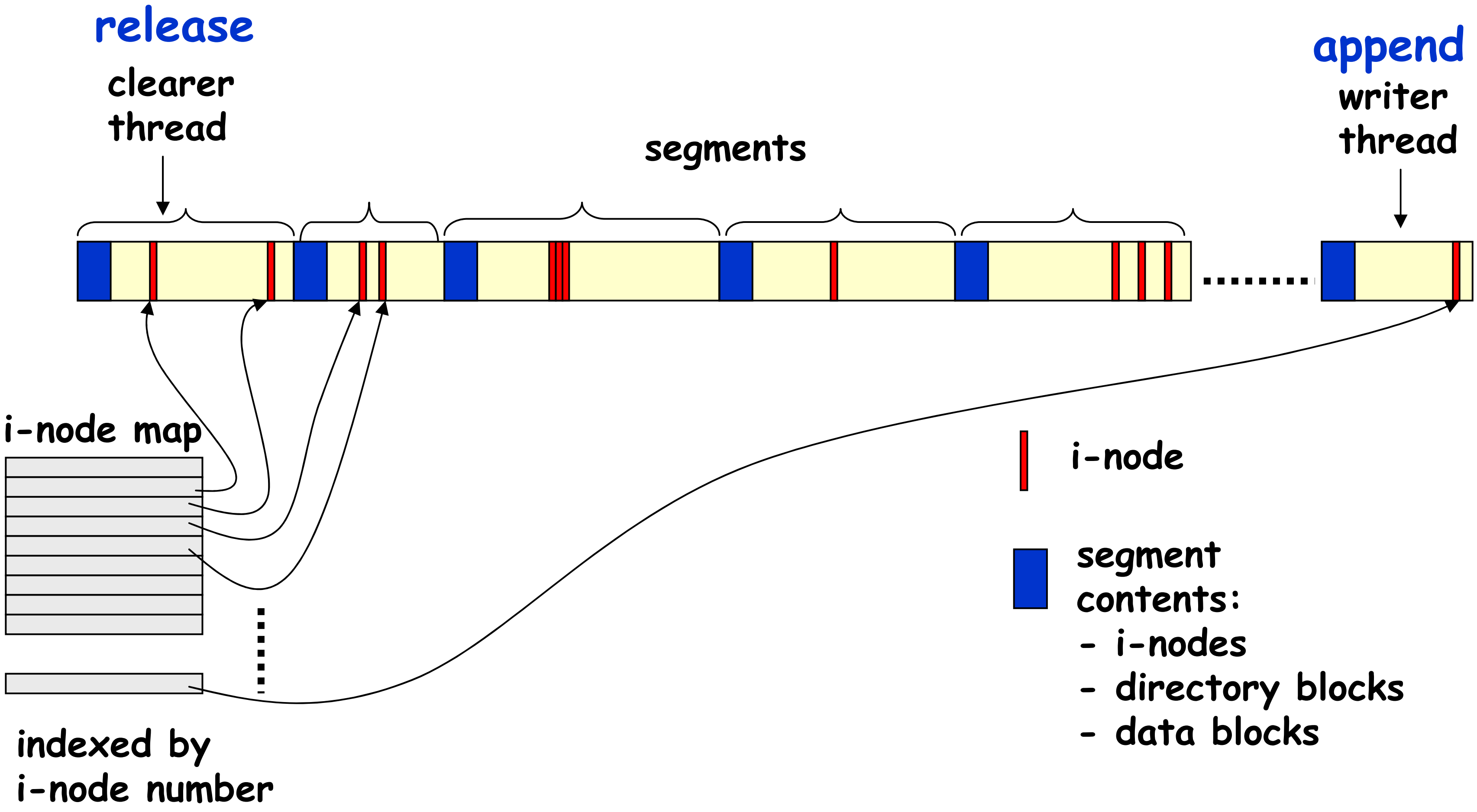
➡ read acceses can be optimized through caching.

➡ write accesses will be the most frequent operation.

➡ write acces to disk becomes a substantial bottleneck.

idea: collect all changes to disk blocks and write them in a single segment to disk.
The resulting data structure is called a "log".

Log File System (LFS) structure



🟥 i-node

🟦 segment contents:
- i-nodes
- directory blocks
- data blocks

Log File System (LFS) structure

- segments are written periodically or on demand
- more overhead for finding information
- much better performance than regular UNIX file system on writing small amounts of data
- better or similar as ordinary UNIX file system for reads and writing large portions of data

Example: Unix file system

Unix supports file names up to 255 characters (previously 14 chars.)

- Files is a sequence of bytes.
- File extensions are conventions.
- Few file types are supported via file type.

Unix supported file types:

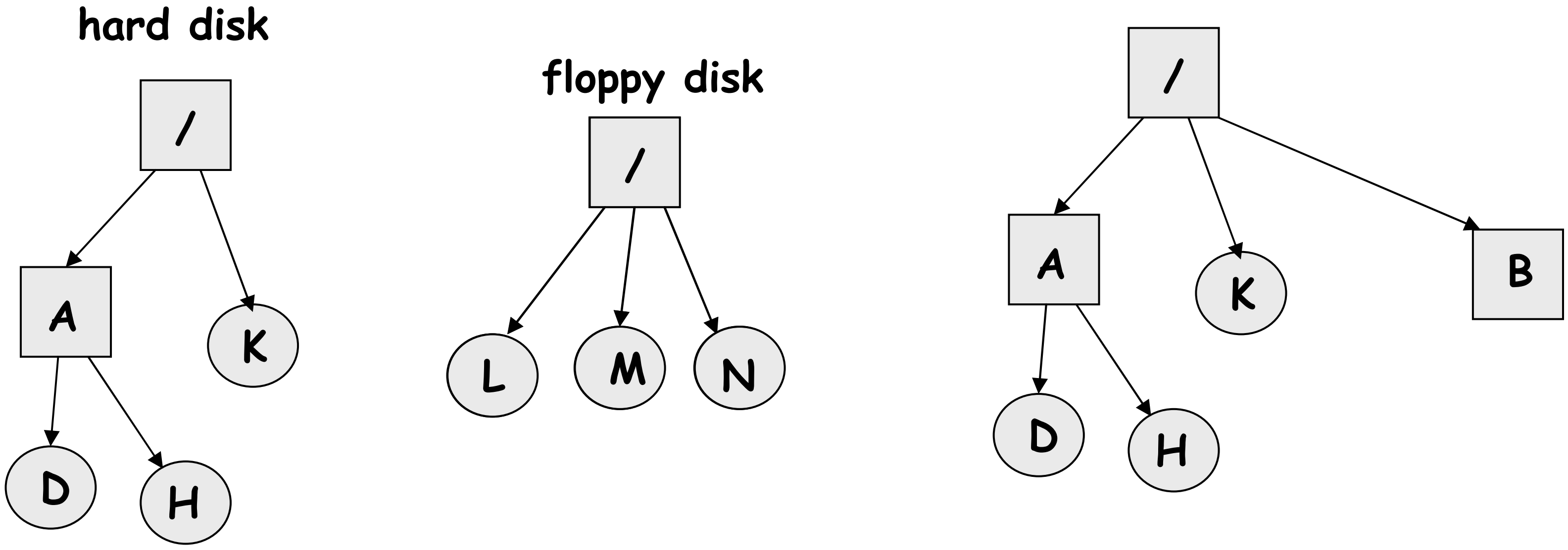
- regular files
- directories contains a list of file names and the resp. i-nodes
- named pipes
- character oriented special files used to model serial I/O devices
- block-oriented special files used to model raw disk partitions

Unix file system: navigating in directories

example dialogue in Unix: typed commands, response

```
cd /
pwd
/
ls
bin  boot  dev  etc  home  lib  lost+found  tmp  usr  var
cd
pwd
/usr/kaiser
ls -all
drwxr-xr-x  14 kaiser  root    4096 March 22 18:17    .
drwxr-xr-x   3 root    root    4096 Dec    11 2003    ..
-rw-------   1 kaiser usr    742068 Nov    13 2004    pubsub-12112003.tar.gz
....
....
cd ..
pwd
/usr
```

Mounting file systems

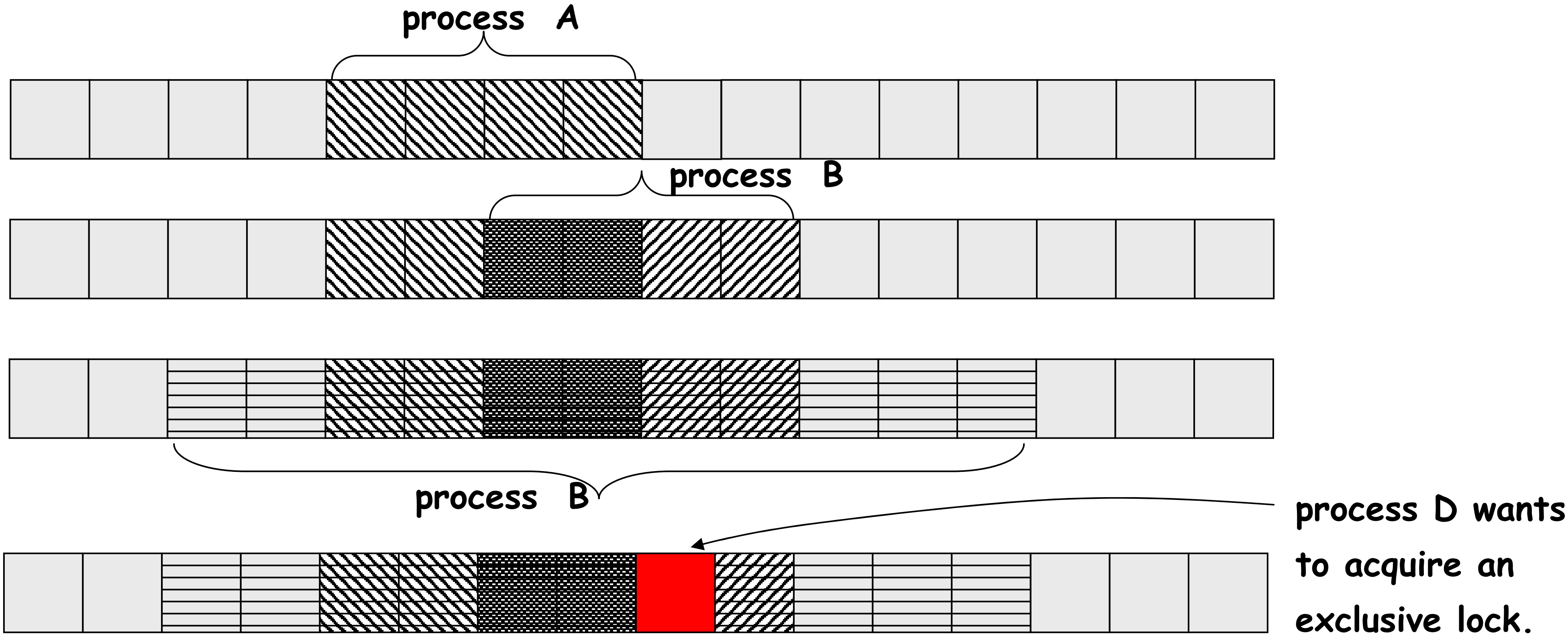


Unix allows a transparent view on different file systems of different storage devices via the mount concept.

Locking file regions

Objective: Improving the granularity of locking down to the byte of a file.

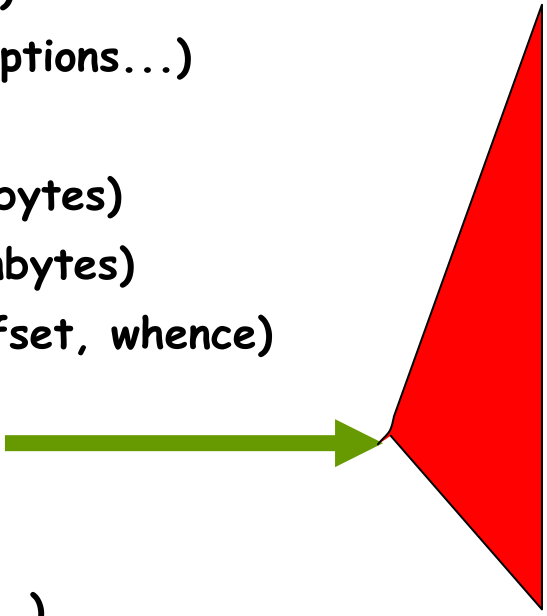
1. shared Locks
2. exclusive locks



Unix system calls

File related system calls

```
fd = creat(name, mode)
fd = open(path, how, options...)
s = close(fd)
n = read(fd, buffer, nbytes)
n = write(fd, buffer, nbytes)
position = lseek(Fd, offset, whence)
s = stat(name, &buf)
s = fstat(fd, &buf)
s = pipe(&fd[0])
s = fcntl(fd, cmd, . . . )
```



- device which holds the file
- i-node number
- mode
- number of links
- group
- size in bytes
- time of creation
- time of last access
- time of last modification

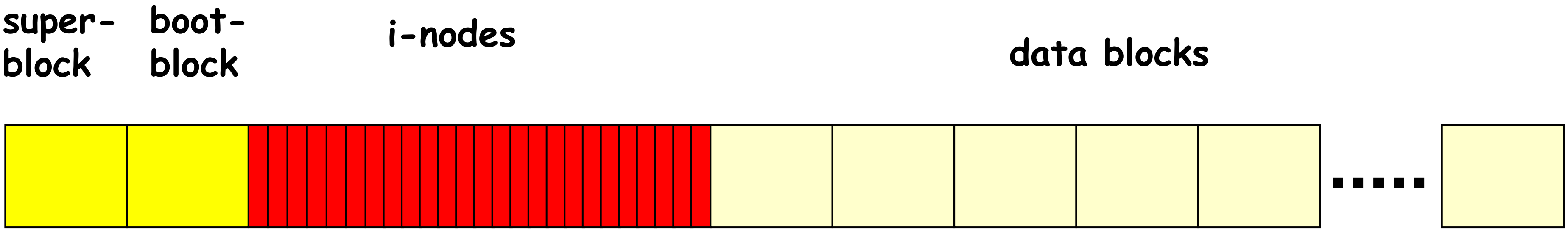
Unix system calls

Directory related system calls

s = mkdir(path, mode)	Create a directory
s = mkdir(path)	delete directory
s = link(oldpath, newpath)	create a link to an exicting file
s = unlink(path)	delete link
s = chdir(path)	change working directory
dir = opendir(path)	open directory for read
s = closedir(dir)	close directory
dirent = readdir(dir)	read a directory entry
rewind(dir) =	rewind und read again

Unix file system management

- Classical Unix System



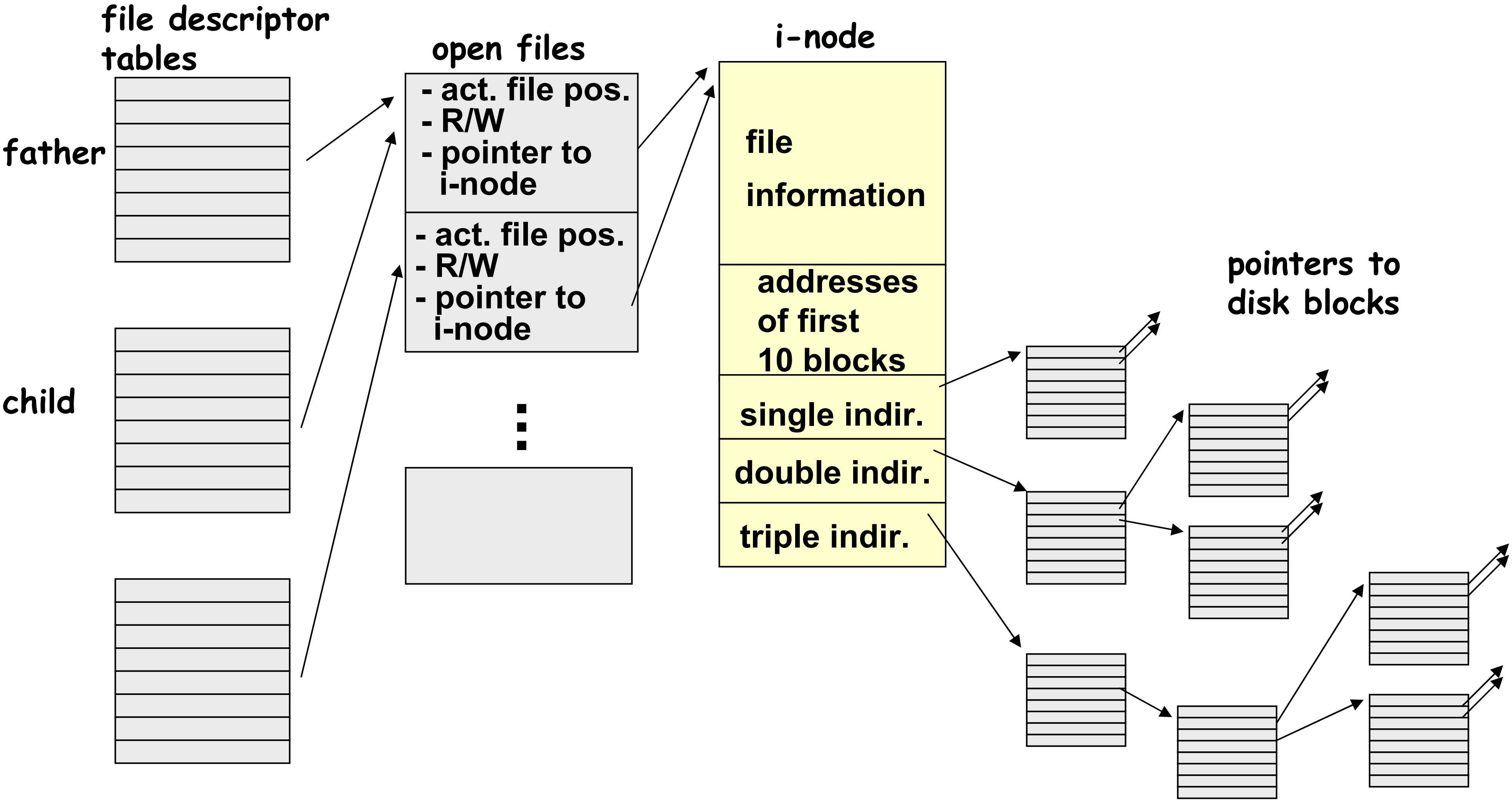
- Berkeley Fast File :
  - long names (255 charakters)
  - structuring the disk in cylinder groups each with own super block, i-nodes and data blocks
  - 2 block sizes

- Linux File System: very similar to Berkeley fast file system.

i-nodes in UNIX

File Mode:	16-Bit Flag which stores access rights
	0 .. 2 rights for "all" users <read, write, exec>
	3 .. 5 rights for the "group" <read, write, exec>
	6 .. 8 rights for "owner" <read, write, exec>
	9 ..11 execution flag
	12..14 file type (regular, char./block-oriented, FIFO pipe)
Link Counter	number of directory references to this i-node
UID	Owner ID
GID	Group ID
Size	in Bytes
File address	39 byte file address information
Last access	date/time
Change of i-node	date/time
Address info for blocks	direct, single ind., double ind., triple ind.

file allocation



capacity of UNIX file

direct	10 blocks	10 K
single indir.	256 blocks	256 K
double ind.	64K blocks	64 M
triple ind.	256x64K blocks	16 G

Windows 2k File System

Windows 2k supports 3 File Systems for compatbility reasons:

FAT 16	(partitions ≤ 2G)
FAT 32	
NTFS	(NT File System)

useful website: http://linux-ntfs.sourceforge.net/ntfs/index.html

main features of NTFS

- ☆ Recoverability after system crashes (including fault-tolerance features)
- ☆ Protection and security
- ☆ Very large disks and very large files
- ☆ Multiple datastreams (which can be addressed under a single file name)
- ☆ General indexing possibilities (acc. to file attributes)

main features of NTFS files

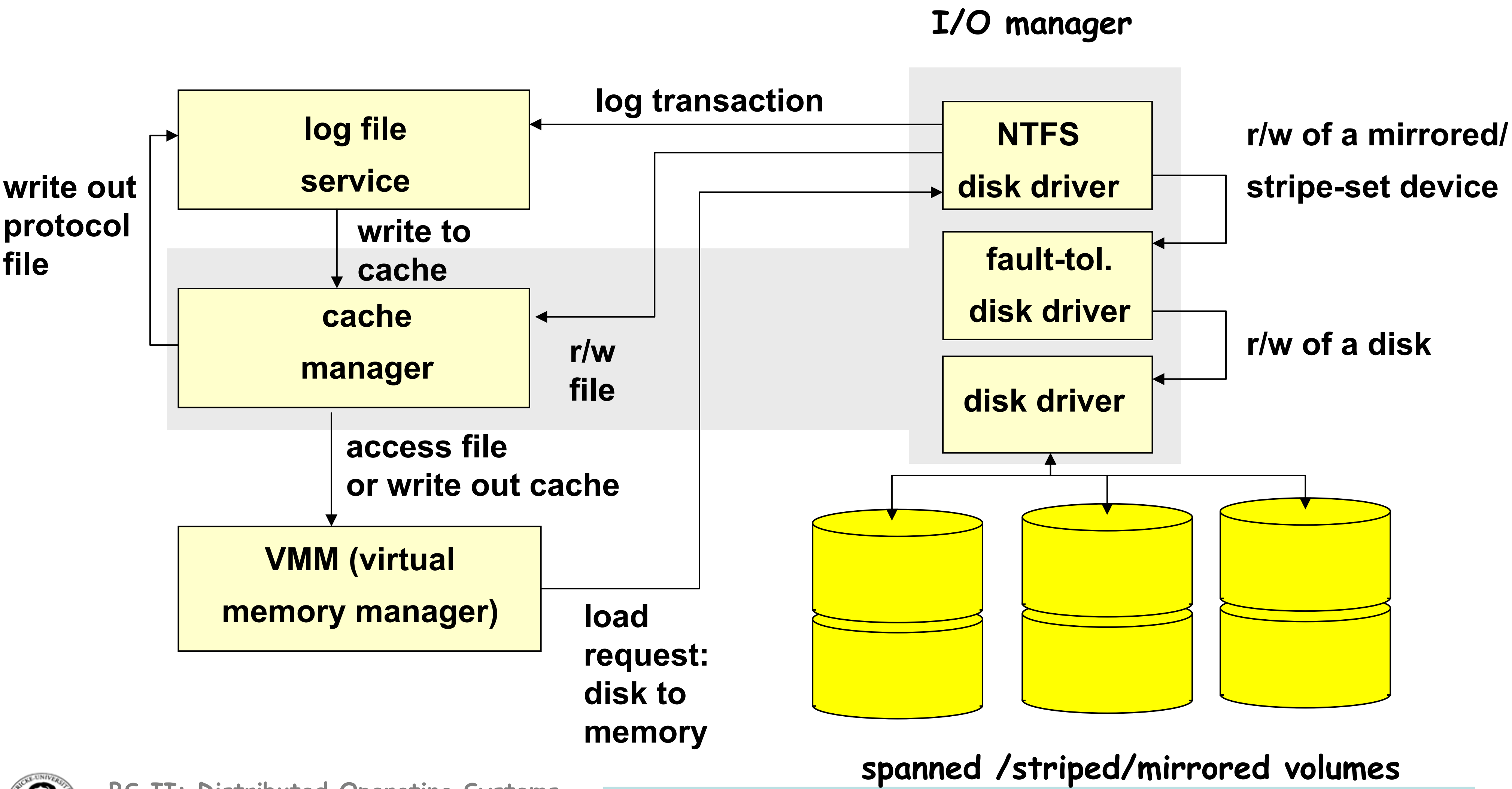
NTFS supports sophisticated naming of files

- long (255 character) file names
- pathnames up to 32767 characters
- unicode representation

NTFS files are not simple byte streams, but..

- comprise multiple byte streams (compatibility with Apple Macintosh FS)
- structured by attributes
- attributes represented by byte streams
- max stream length: 2^{64} bytes (18,4 Exabytes)

W2K components supporting NTFS



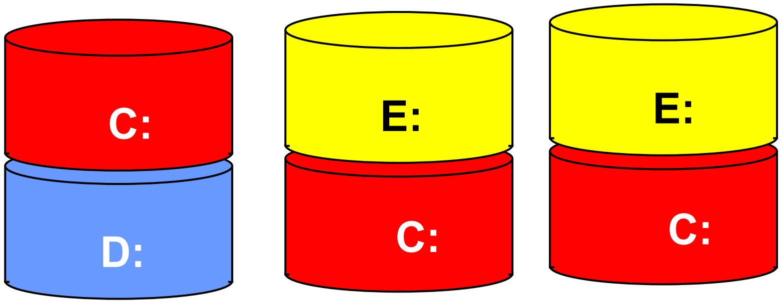
Spanned Volumes:

Logical partitions span multiple physical disks.

Motivations:

Transparent extensibility

Concurrent access to multiple physical disks improves performance



Striped Volumes:

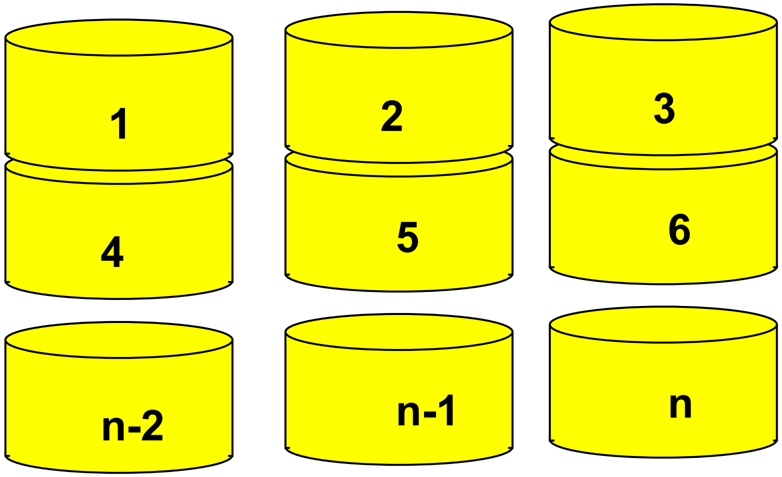
A physical disk drive includes multiple disks. Appears as a single disk for the operating system with improved performance and reliability.

Motivations:

Concurrent access to multiple physical disks improves performance.

Redundant Array of (inexpensive) independent disks for FT.

(RAID-1, RAID-5)



Win32 API

Important API functions for files:

Win32	Unix	
CreateFile	open	Create or open a file; Return a handle
DeleteFile	unlink	Delete a file
CloseHandle	close	Close a file
ReadFile	read	Read data from file
WriteFile	write	Write data to file
SetFilePointer	lseek	position read pointer
GetFileAttributes	stat	Get File Attributes
LockFile	fcntl	Lock part of a file for multiple access
UnlockFile	fcntl	Release lock

Win32 API

Important API functions for directories:

Win32	Unix	
CreateDirectory	mkdir	Create a directory
RemoveDirectorY	unlink	Delete empty directory
FindFirstFile	opendir	Open directory and read entries
FindNextFile	readdir	Read next entry
MoveFile	rename	move file in another directory
SetCurrentDirectory	chdir	cange current working directory

NTFS basic concepts

Volume and File structure

Volume: Logical disk partition

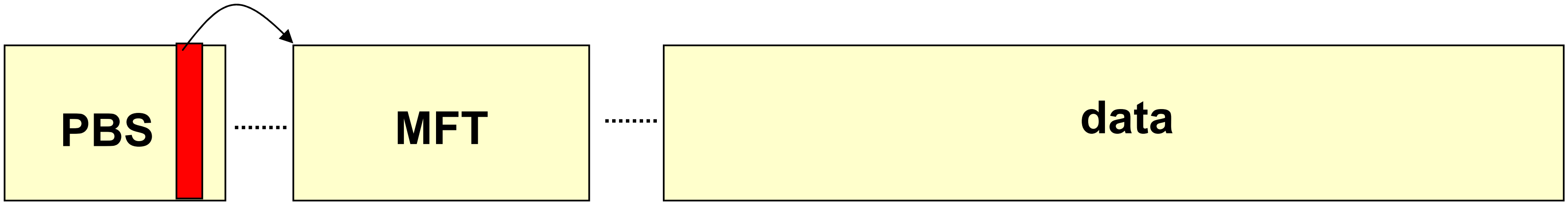
Sector: Smallest physical storage unit (most common size: 512 Byte)

Cluster: One or more consecutive sectors of the same track (corresp. to a block)

The Cluster is the basic unit of storage allocation in NTFS

Volume size	sector/cluster	cluster size
≤ 512 MB	1	512 Byte
512 MB - 1G	2	1 K
1-2 G	4	2 K
2-4 G	8	4 K
4-8 G	16	8 K
⋮	⋮	⋮
> 32 G	128	64K

NTFS volume structure



- PBS: Partition Boot Sector (up to 16 sectors)

- MFT: Master File Table
  - is a file that can be placed freely
  - contains meta data: MFT2, Log file, Cluster bit map,
    Attribute definition table, . . .
  - \+ user file descriptors

Structure of NTFS

Master

File

Table

1 kB

:

16	1. user file	
15	reserved for future use	
14	reserved for future use	
13	reserved for future use	
12	reserved for future use	
11	$Extend	Extensions, Quotas etc.
10	$Upcase	Conversion table for uper/lower case letters
9	$Secure	Security Information
8	$BadClus	List of bad blocks
7	$Boot	Bootstrap Loader
6	$Bitmap	Bitmap of used/free blocks
5	$	Root directory
4	$AttrDef	Attribute specification
3	$Volume	Volume file
2	$LogFile	Log file for recovery
1	$MftMirr	Copy of MFT
0	$MFT	Master File Table

files holding meta-data (0–11)

Structure of NTFS

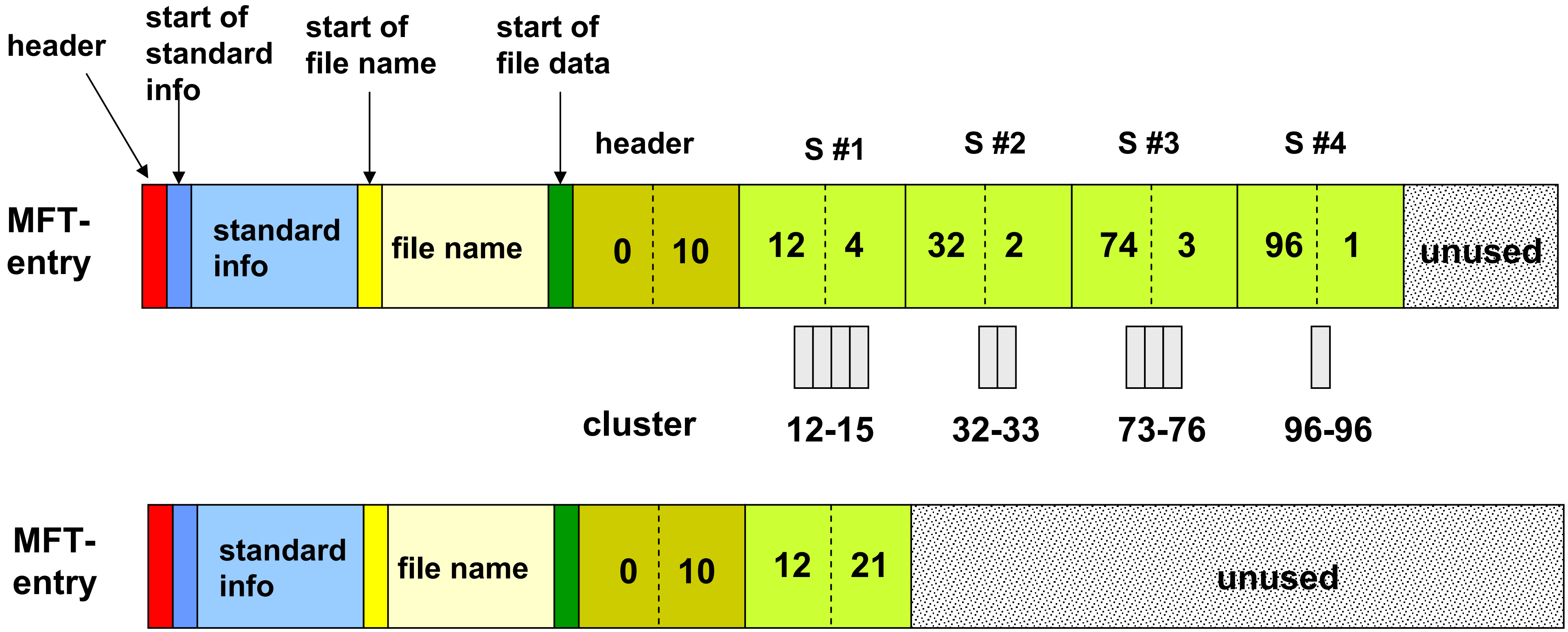
Usual attributes in MFT entries:

Default Information	Owner, protection info, time stamp, link counter, etc....
File name	file name in unicode
Security descriptor	(old) information now in $Extend and $Secure fields
Attribute list	Place wher additional MFT entries are stored if required
Object-ID	64 Bit file ID for internal use (unique for a volume)
Reparse	used for creating symbolic links
Volume name	used in $volume only
Volume attribute	used in $volume only
Index root	used for directories (called index in Microsoft terminology)
Index allocation	used for very large directories
Bitmaps	used for very large directories
Logging-support system	controls the logging in the log file
data	data stream

Structure of NTFS

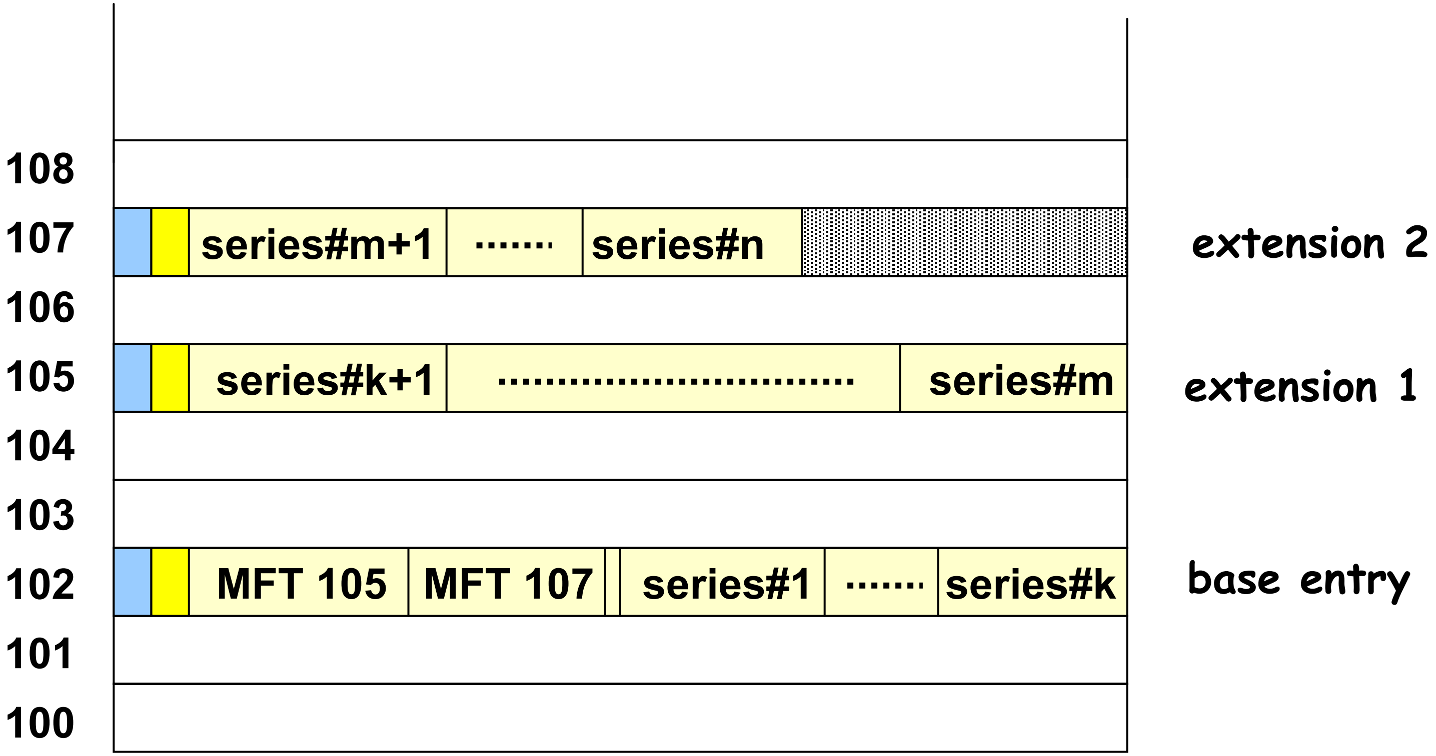
The "data" attribute: obviously, not all data fit in a single entry.

Problem: How to find the associated blocks (clusters)?



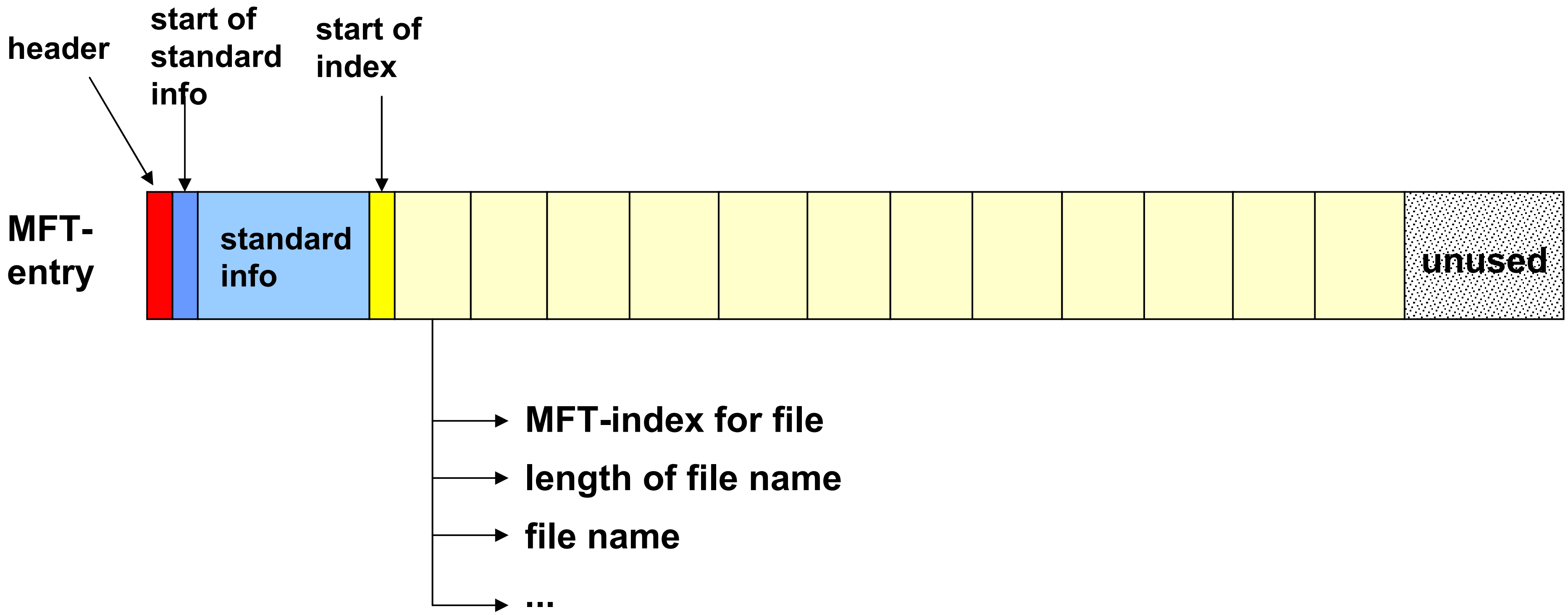
Structure of NTFS

Storing file clusters in multiple MFT entries



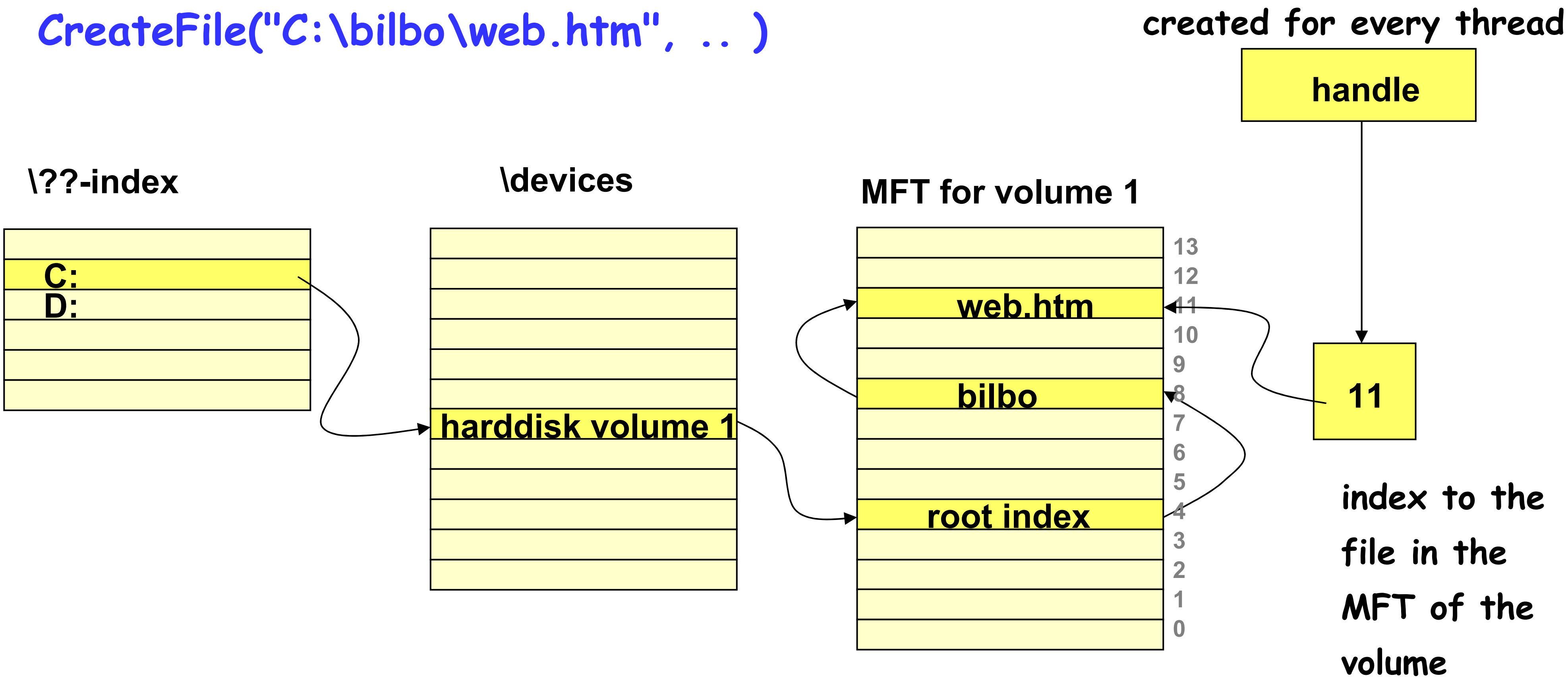
Structure of NTFS

MFT entry for a small index (directory)



Structure of NTFS

CreateFile("C:\bilbo\web.htm", ..)



created for every thread

handle

\??-index

C:
D:

\devices

harddisk volume 1

MFT for volume 1

web.htm

bilbo

root index

13
12
11
10
9
8
7
6
5
4
3
2
1
0

11

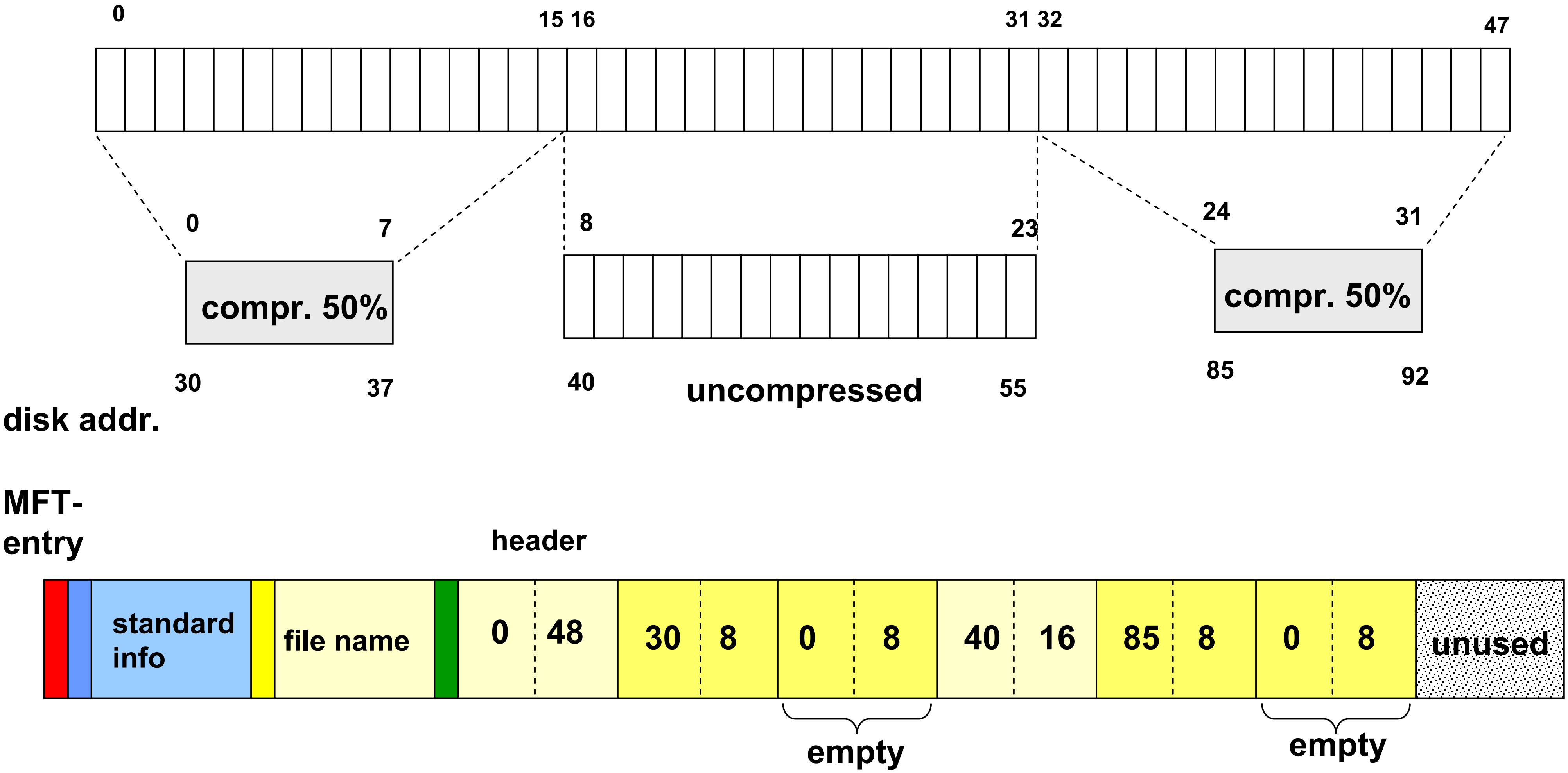
index to the file in the MFT of the volume

Structure of NTFS

More features:

- Compression of Addresses (16 --> 4)
- Compression of files
- Encrypted files
- Security and Access control

Compression of files

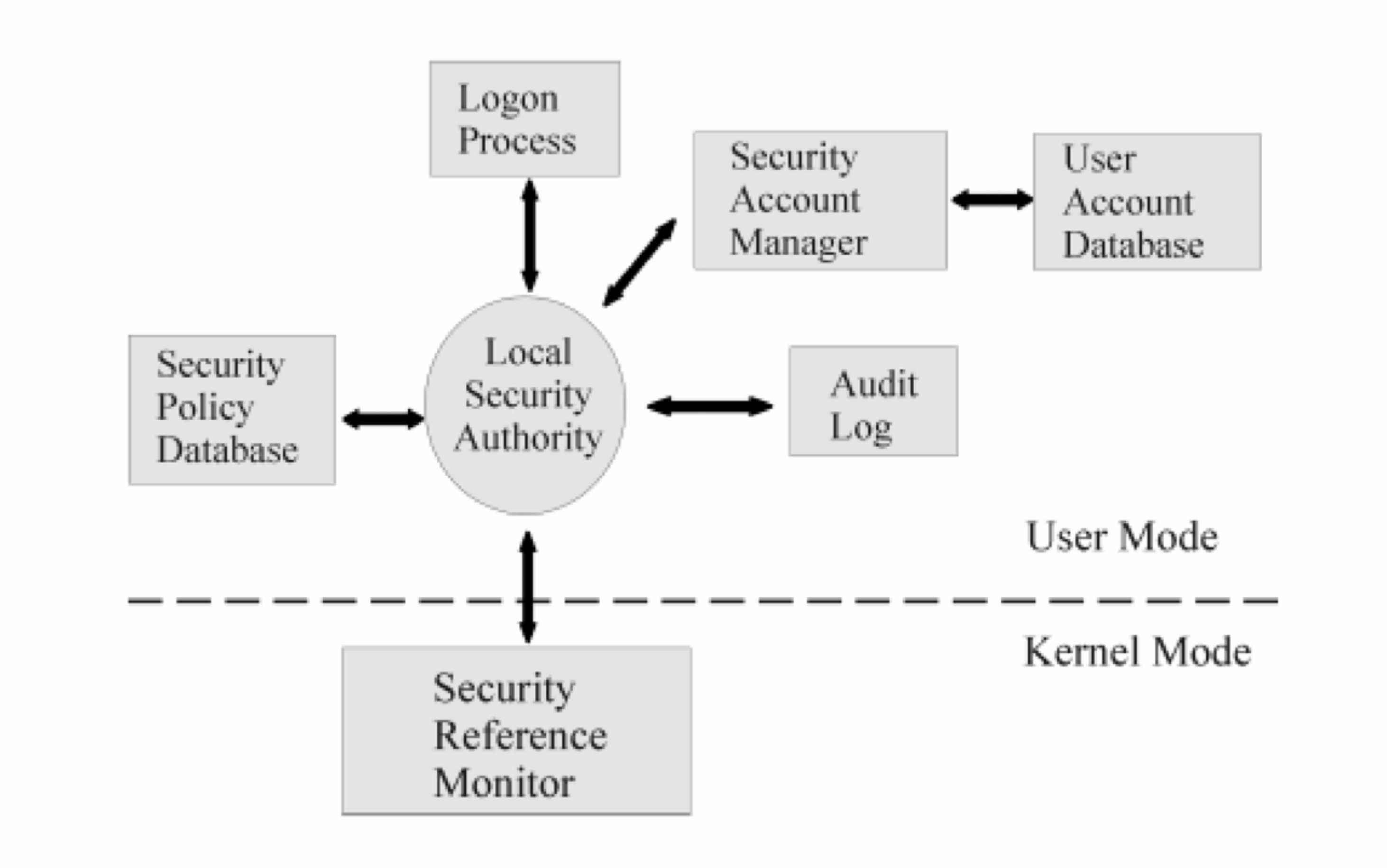


Security and access protection

- secure login and antispoofing
- discretionary access control
- privileged access control
- process address space protection
- prevention of data leaks by zeroing all new pages before loading
- security auditing

overall NT security model

http://www.ciac.org/ciac/documents/CIAC-2317_Windows_NT_Managers_Guide.pdf



NT logon process

Windows NT logon processes provide mandatory logon for user identification and cannot be disabled.

To protect against spoofing, the logon process begins with a Welcome message box that requests the user to press Ctrl, Alt and Del keys before activating the actual logon screen.

Logon

LSA Authentication Package run

Verify Authentication in SAM Database

SAM returns SID(s)

LSA creates Access Token

LSA creates Subject

the access token

header	expir. time	groups	standard DACL	owner SID	group SID	restricted SIDs	privileges

Security ID (SID): The SID is a variable length unique name (alphanumeric character string) that is used to identify an object, such as a user or a group of users in a network of NT/2000 systems.

Expiration time: defines validity interval for the access token (currently not used)

Discretionary Access Control List (D ACL): Default ACL when they are created by a process and no other ACL is specified.

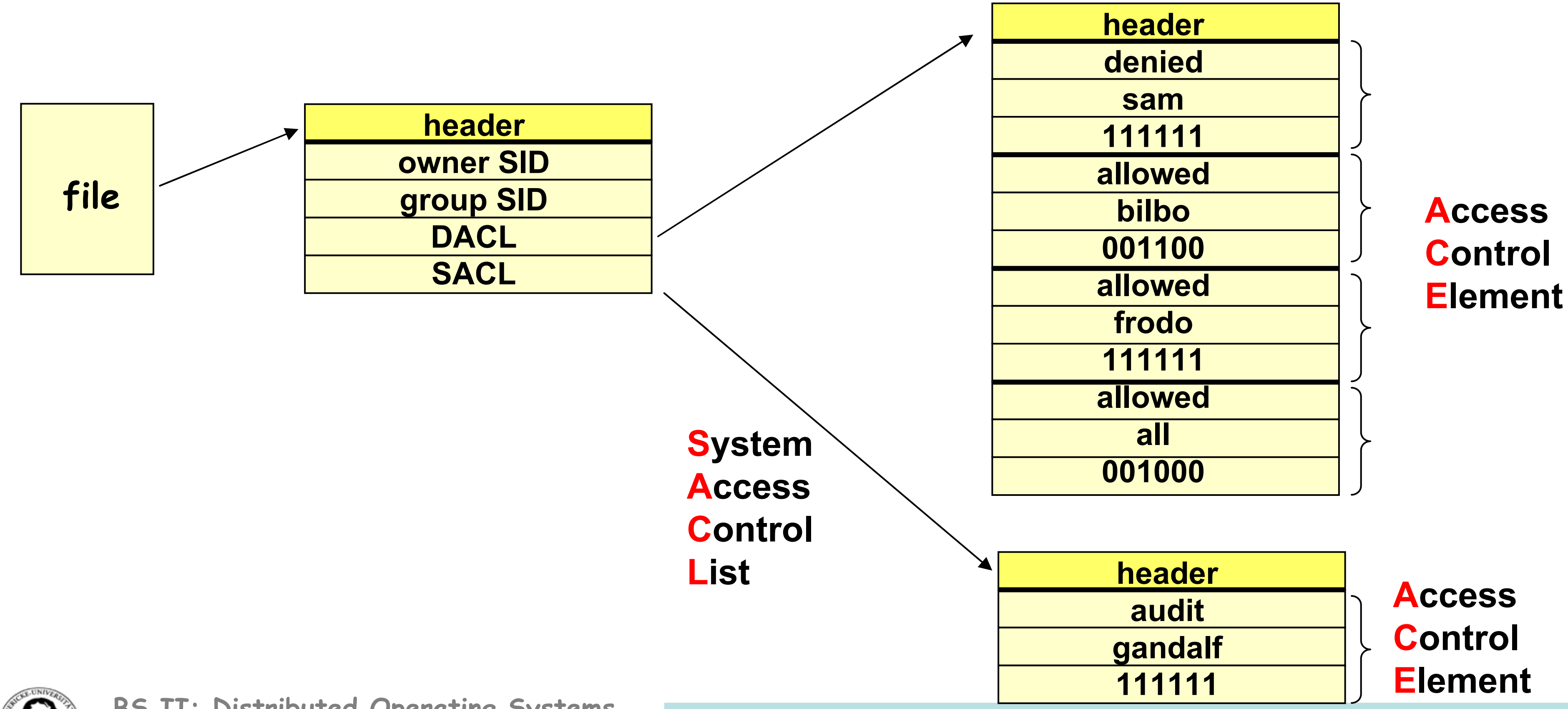
Owner/group SID: indicates the user/group who owns the process.

Restricted SID: enables the cooperation of trusted and non-trusted processes by contraining access for the latter.

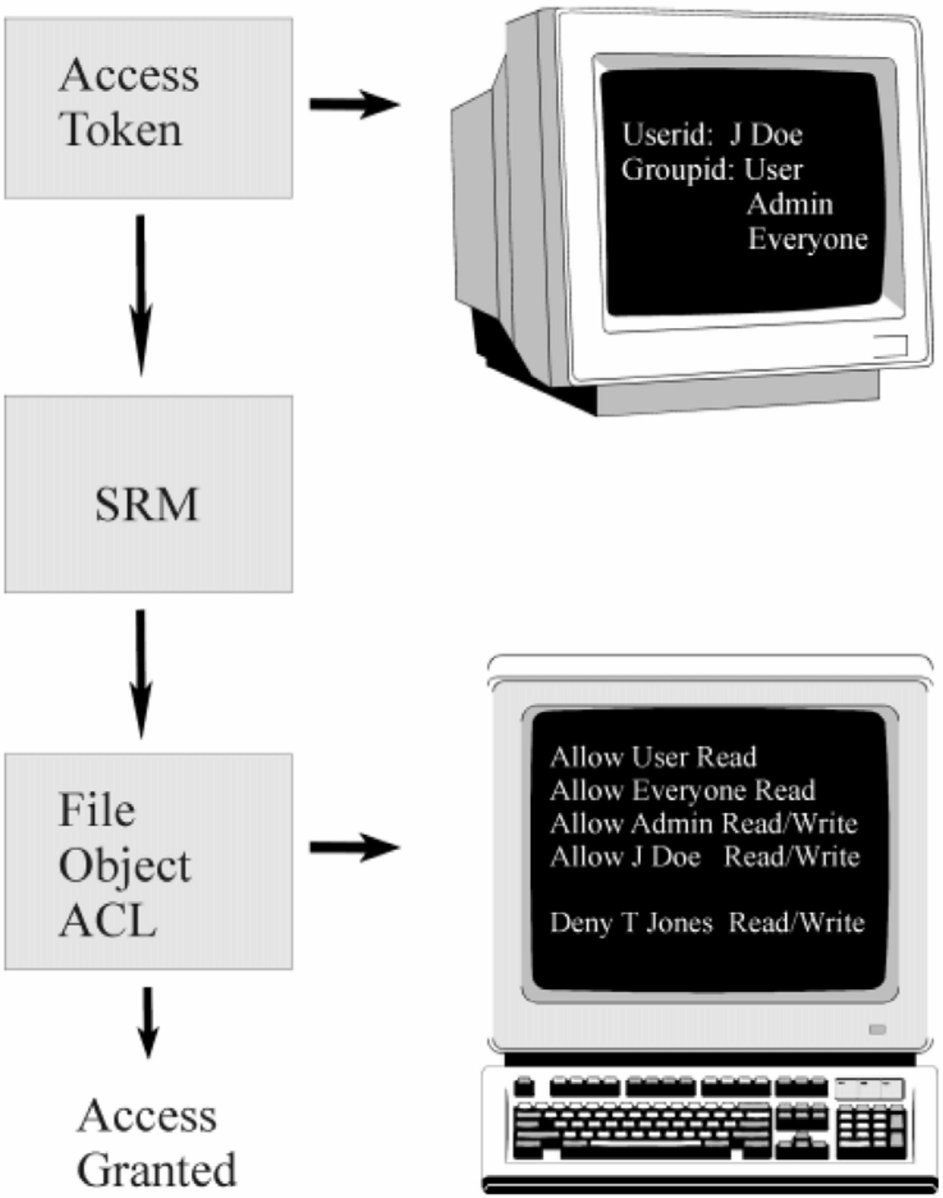
Privileges: enable to define "admin rights" in a more fine-grained fashion and associate these with user processes.

the security descriptor

- is associated with every object
- defines who may access the object with which operation



SRM access validation



Access Token

Userid: J Doe
Groupid: User
Admin
Everyone

SRM

File Object ACL

Allow User Read
Allow Everyone Read
Allow Admin Read/Write
Allow J Doe Read/Write

Deny T Jones Read/Write

Access Granted