Imperial College London

Operating Systems

File Systems

Course 211
Spring Term 2016-2017

Based on slides by Daniel Rueckert

Peter Pietzuch

prp@doc.ic.ac.uk
http://www.doc.ic.ac.uk/~prp

File Systems Objectives

Long term, nonvolatile, online storage

- e.g. programs, data, text, ...

Sharing of information or software

- e.g. editors, compilers, applications, ...

Concurrent access to shared data

- e.g. airline reservation system, ...

Organisation and management of data

- e.g. convenient use of directories, symbolic names, ...
- e.g. automatic backups, snapshots, ...

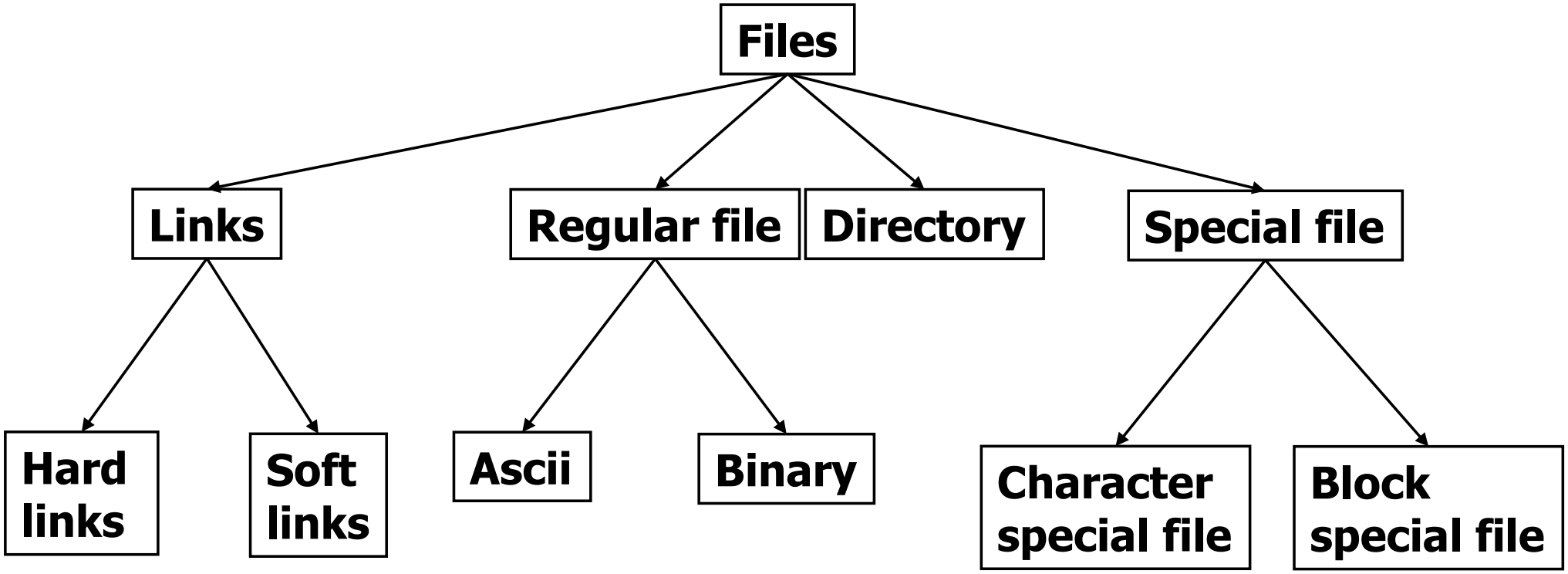
File: Named collection of data of arbitrary size

File Naming

file type	usual extension	function
executable	exe, com, bin or none	read to run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
print or view	arc, zip, tar	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information

Typical file extensions

File Types


Files
Links
Regular file
Directory
Special file
Hard links
Soft links
Ascii
Binary
Character special file
Block special file

File User Functions

Create:	Create empty file Allocate space and add to directory
Delete:	Deallocate space Invalidate or remove directory entry
Open:	Search directory for file name Check access validity and set pointers to file
Close:	Remove pointers to file
Read:	Access file, update current position pointers
Write:	Access file, update pointers
Reposition/seek:	Set current position to given value
Truncate:	Erase contents but keep all other attributes
Rename:	Change file name
Read attributes:	e.g. creation date, size, archive flag, ...
Write attributes:	e.g. protection, immutable flag, ...

Unix/Linux: File System Calls

System Call	Description
`fd = open (file, how, ...)`	Open a file for reading/writing
`s = close (fd)`	Closing an open file
`n = read (fd, buffer, nbytes)`	Read data from file to buffer
`n = write (fd, buffer, nbytes)`	Write data from buffer to file
`position = lseek (fd, offset, ..)`	Move file pointer
`s = stat(name, &buf)`	Get file's meta-data
`s = fnctl (fd, cmd, ...)`	File locking and other operations

FS Support Functions

Logical name to physical disk address translation

- i.e. `/home/prp/.email` → disk 2, block 493

Management of disk space

- Allocation and deallocation

File locking for exclusive access

Performance optimisation

- Caching and buffering

Protection against system failure

- Back-up and restore

Security

- Protection against unauthorised access (Security)

File Attributes I

File attributes may be held within given directory system

Basic information

file name:	symbolic name; unique within directory
file type:	text, binary, executable, directory, ...
file organisation:	sequential, random, ...
file creator:	program which created file

Address information

volume:	disk drive, partition
start address:	(cyl, head, sect), LBA
size used	
size allocated	

File Attributes II

Access control information

owner: person who controls file (often creator)

authentication: password

permitted actions: read, write, delete for owner/others

Usage information

creation timestamp: date and time

last modified: could include user id

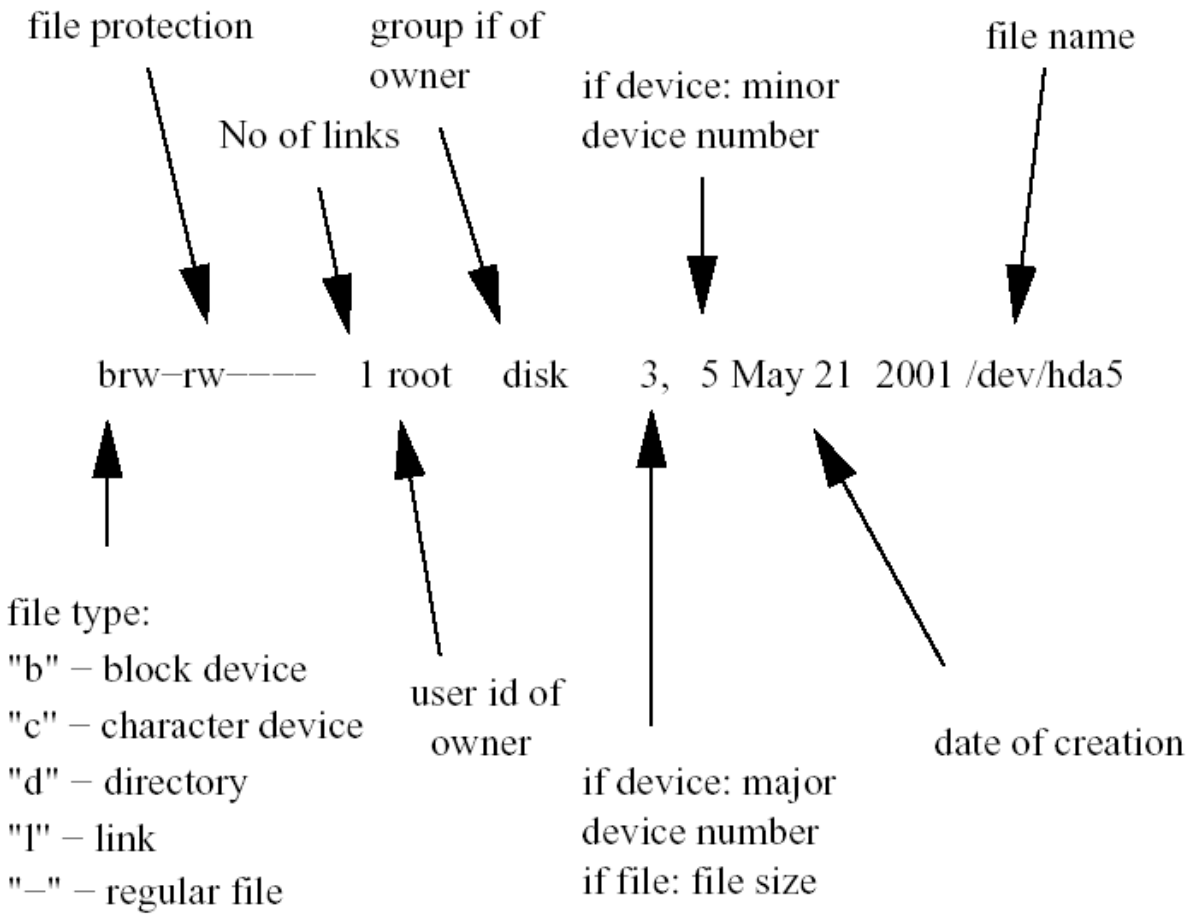
last read

last archived

expiry date: when file will be automatically deleted

access activity counts: number of reads/writes

Unix/Linux: File Attributes



Unix/Linux: stat System Call I

File attributes can be accessed using system call **`stat(2)`**

- Returns information about specified file in **`struct stat`**
- Try: **`man 2 stat`**

```
void struct stat {
        /* device */
        dev_t st_dev;
        /* inode */
        ino_t st_ino;
        /* protection */
        mode_t st_mode;
        /* number of hard links */
        nlink_t st_nlink;
        /* user ID of owner */
        uid_t st_uid;
```

Unix/Linux stat System Call II

```
        /* group ID of owner */
        gid_t st_gid;

        /* device type (if inode device) */
        dev_t st_rdev;

        /* total size, in bytes */
        off_t st_size;

        /* blocksize for filesystem I/O */
        unsigned long st_blksize;

        /* number of blocks allocated */
        unsigned long st_blocks;

        /* time of last access */
        time_t st_atime;

        /* time of last modification */
        time_t st_mtime;

        /* time of last change */
        time_t st_ctime;
};
```

File System Organisation

Space Allocation

Dynamic space management

- File size naturally variable
- Space allocated in **blocks** (typically 512 - 8192 bytes)

Choosing block size

- Block size too large ➔ wastes space for small files
  - More memory needed for buffer space
- Block size too small ➔ wastes space for large files
  - High overhead in terms of management data
  - High file transfer time: seek time greater than transfer time

Various methods exist for accessing blocks belonging to file

- **Contiguous file allocation**
- **Block chaining**
- **File allocation table**
- **Index blocks**

Contiguous File Allocation

Place file data at contiguous addresses on storage device

Advantages

- Successive logical records typically physically adjacent

Disadvantages

- External fragmentation
- Poor performance if files grow and shrink over time
- File grows beyond size originally specified and no contiguous free blocks available
  - Must be transferred to new area of adequate size
  - Leads to additional I/O operations

Block Linkage (Chaining) I

When locating data block:

- Chain must be searched from beginning
- If blocks dispersed throughout disk, search process slow
  - Block-to-block seeks occur

Wastes pointer space in each block

Insertion/deletion by modifying pointer in previous block

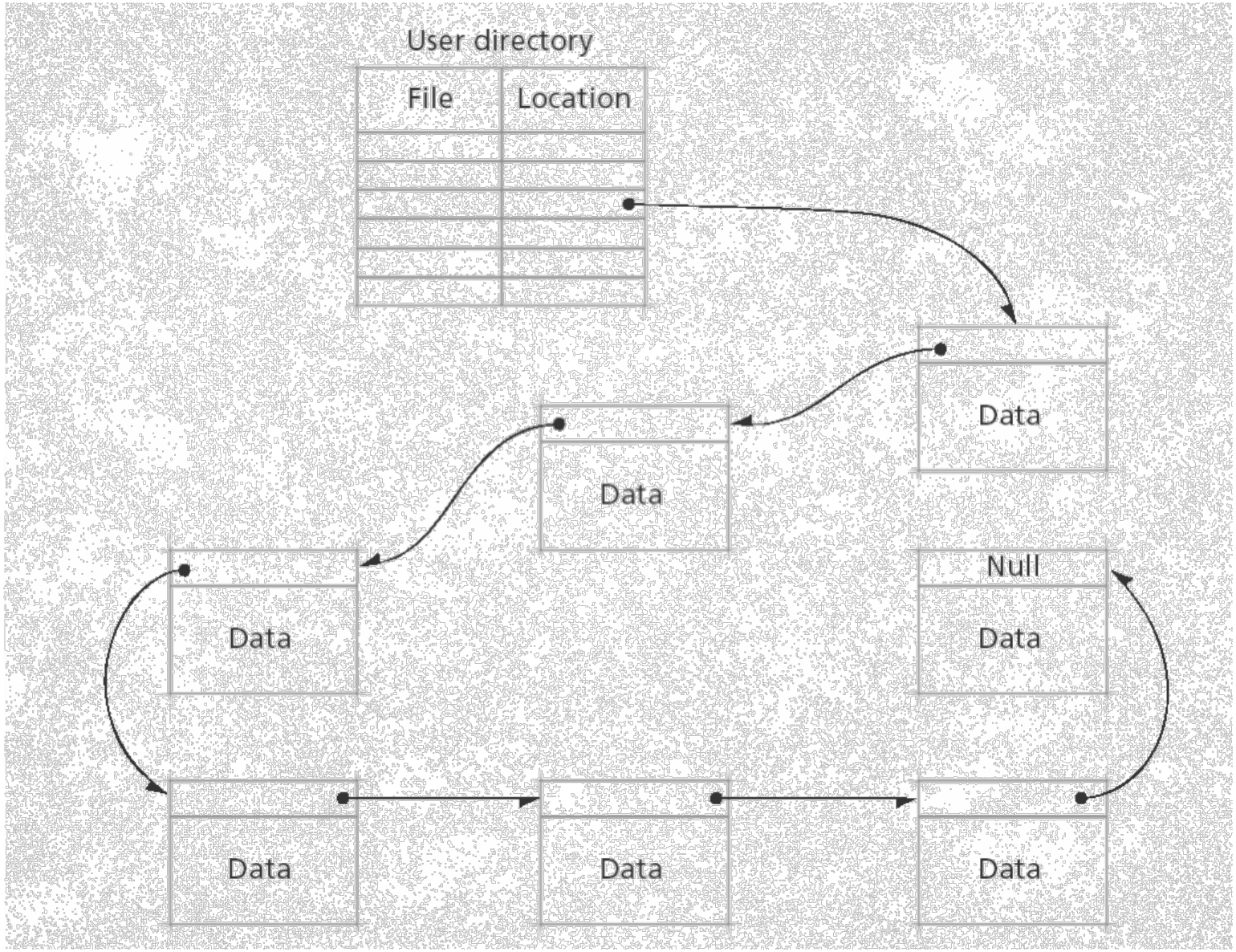
Large block sizes

- Can result in significant internal fragmentation

Small block sizes

- Data spread across multiple blocks, dispersed through disk
- Poor performance due to many seeks

Block Linkage (Chaining) II



Block Allocation Table I

Block allocation table stores pointers to file blocks

- Directory entries indicate first block of file
- Block number as index into block allocation table
  - Determines location of next block
  - If current block = last block, set table entry to **`null`**

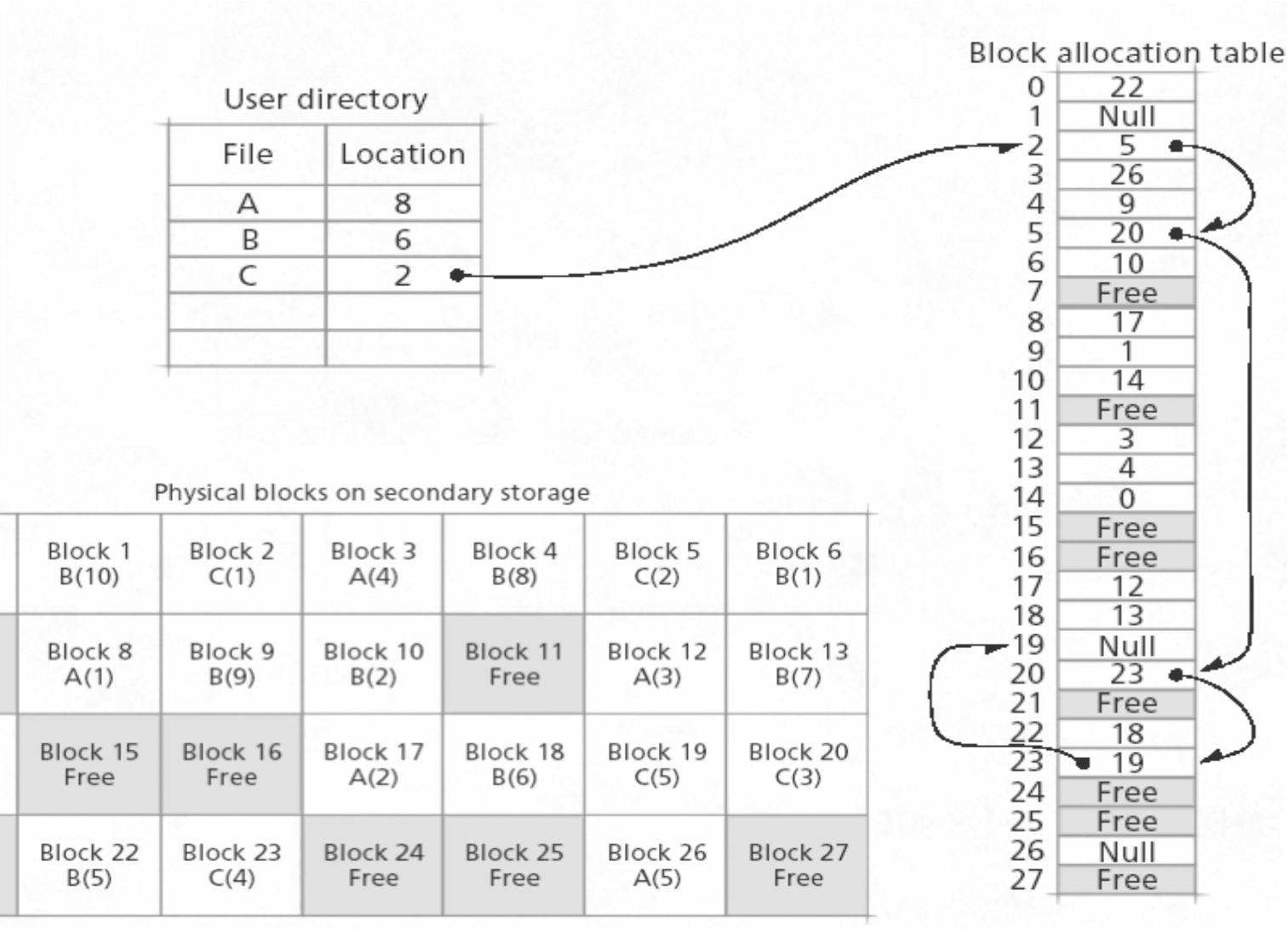
File Allocation Table (e.g. MSDOS/Windows (FAT16/32))

- Stored on disk but cached in memory for performance

Reduces number of lengthy seeks to access given record

- But files become fragmented ➔ periodic defragmentation
- Table can get very large (solution?)

Block Allocation Table II



Example: Block Linkage vs. FAT

Consider a disk with a block size of 1024 bytes. Each disk address can be stored in 4 bytes. Block linkage is used for file storage, i.e. each block contains the address of the next block in the file.

1. How many block reads will be needed to access:
   - the 1022nd data byte?
   - the 510,100th data byte?

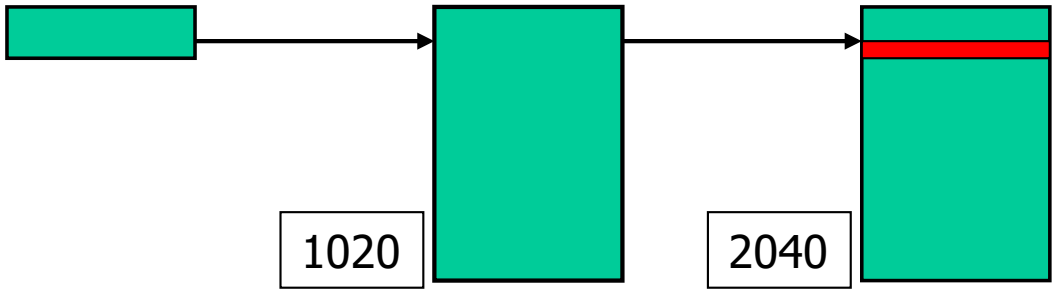
   Note: 500 * 1020 = 510000 and 498 * 1024 = 509952

2. How does this change if a file allocation table (FAT) is used?

Answer: Block Linkage vs. FAT I

Block Linkage: There are 1020 data bytes per block. The 1022nd byte is resident on the 2nd disk block:

- Therefore 2 disk reads are required



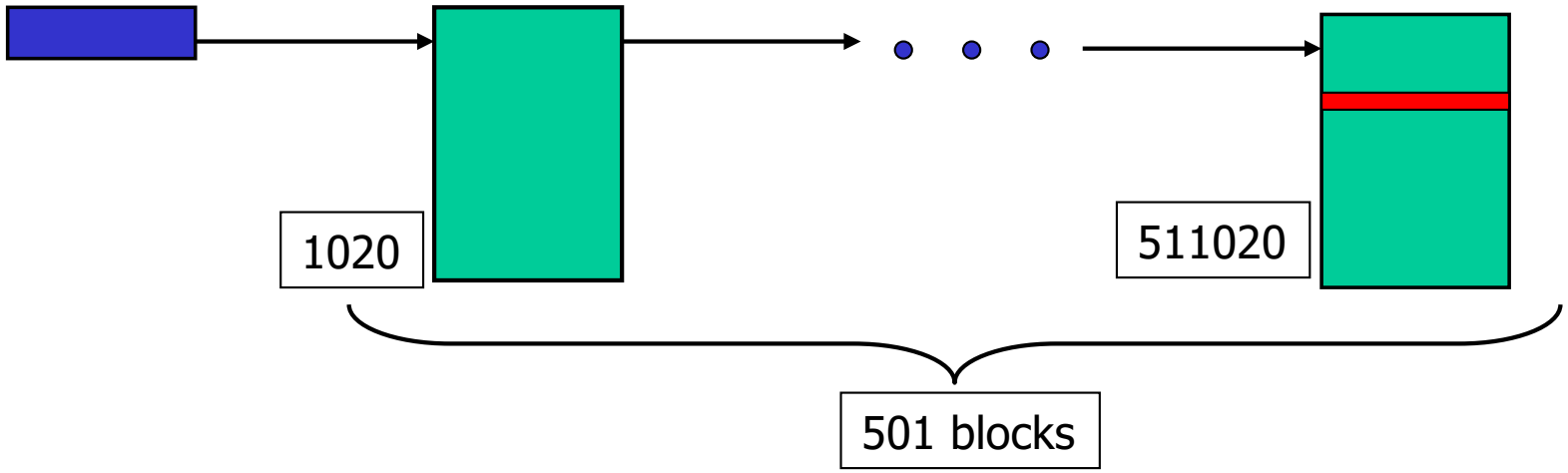
Answer: Block Linkage vs. FAT II

Block Linkage: There are 1020 data bytes per block. The 1022nd byte is resident on the 2nd disk block:

- Therefore 2 disk reads are required

The 510,100th byte is resident in the 501st disk block:

- Therefore 501 disk reads are required



Answer: Block Linkage vs. FAT III

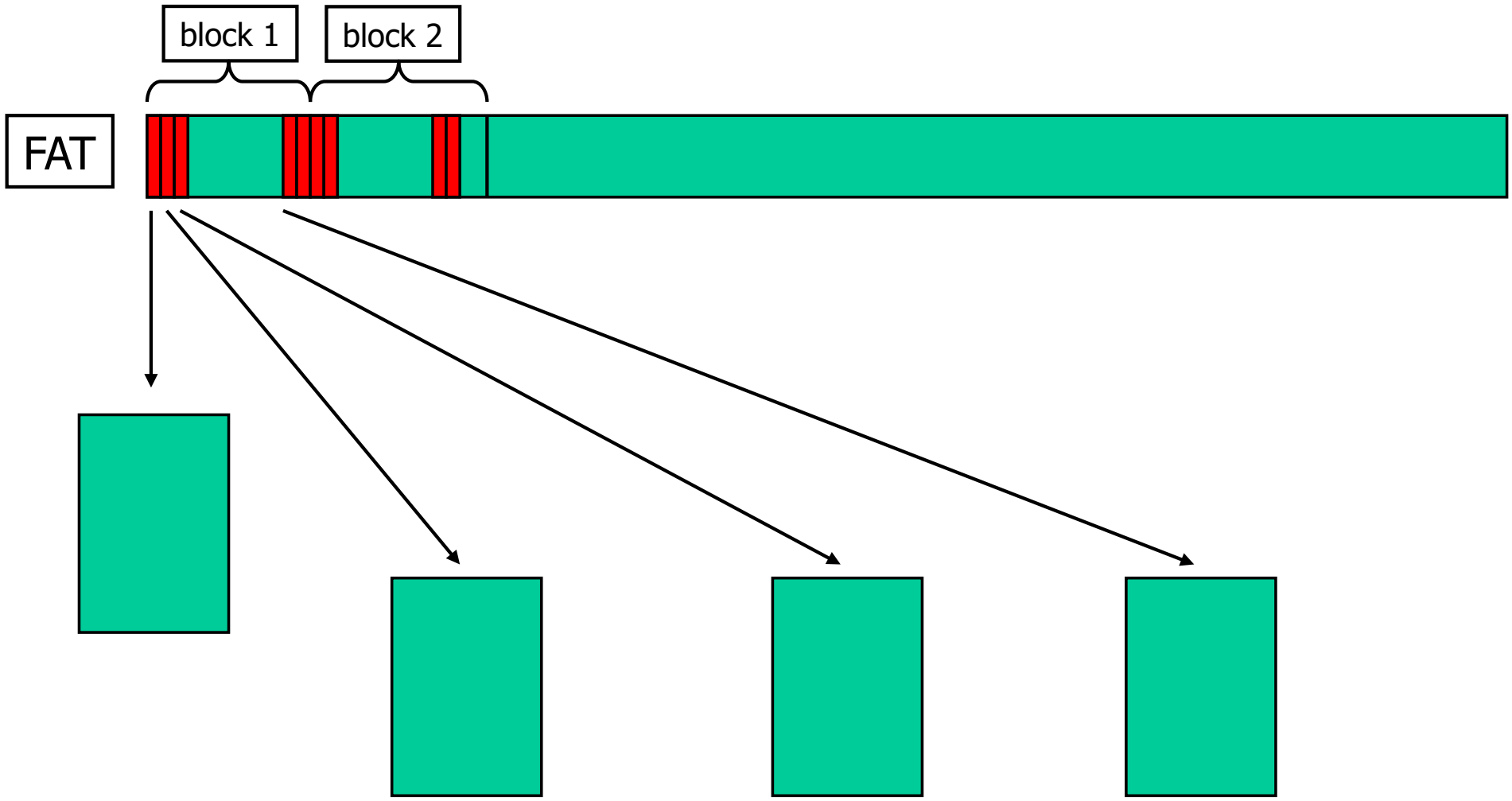
FAT: There are 1024 data bytes per block. Each block of the FAT can represent 1024 / 4= 256 data blocks.

- The 1020th byte is on the 1st block, requires 1 read for the FAT and 1 read for the data block, for a total of 2 reads.

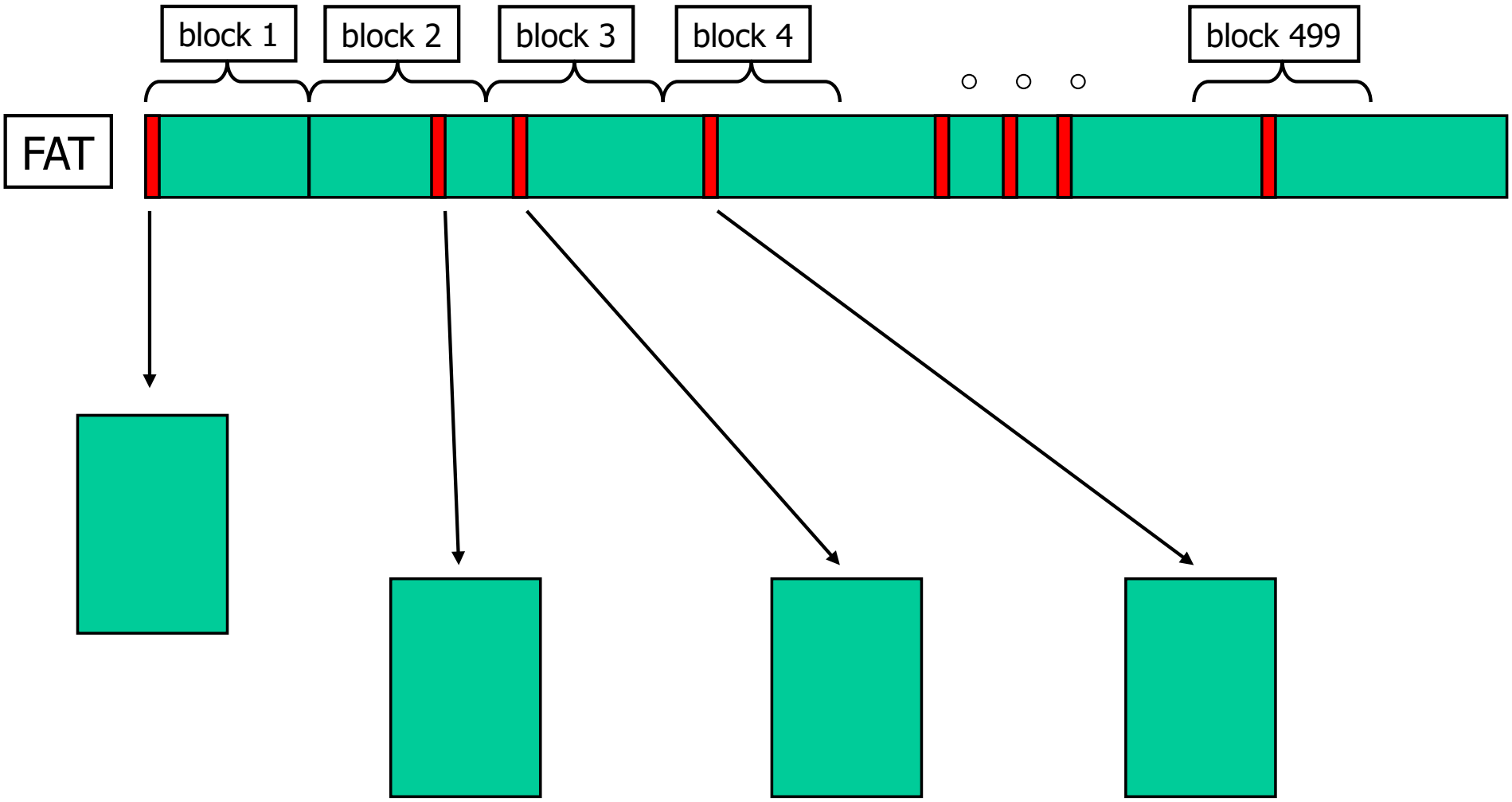
- The 510,100th byte is on the 499th data block:
  - At best, all of the first 499 blocks of the file can be represented in 2 FAT blocks.
  - At worst, 499 reads could be performed for the FAT.
- Either case requires 1 extra read for the data.
  - Therefore the best case requires 3 reads.
  - Therefore the worst case requires 500 reads.

This would benefit from caching parts of the FAT in memory

Answer: Block Linkage vs. FAT IV



Answer: Block Linkage vs. FAT V



Index Blocks I

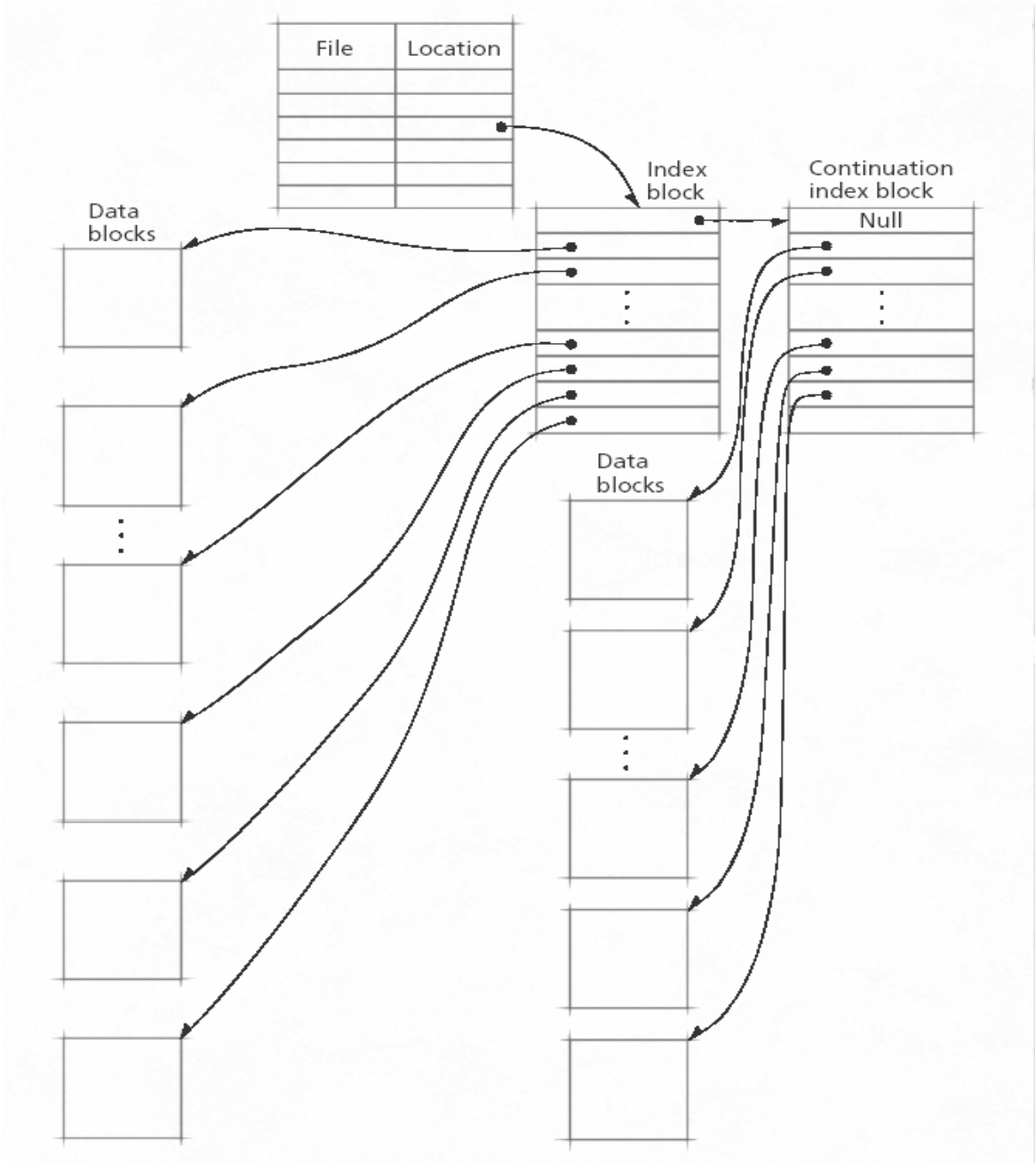
Each file has one (or more) **index blocks**

- Contain list of pointers that point to file data blocks
- File's directory entry points to its index block
- Chaining: May reserve last few entries in index block to store pointers to more index blocks

Advantages over simple linked-list implementations

- Searching may take place in index blocks themselves
- Place index blocks near corresponding data blocks ➔ quick access to data
- Cache index blocks in memory

Index Blocks II



Unix/Linux: Inodes

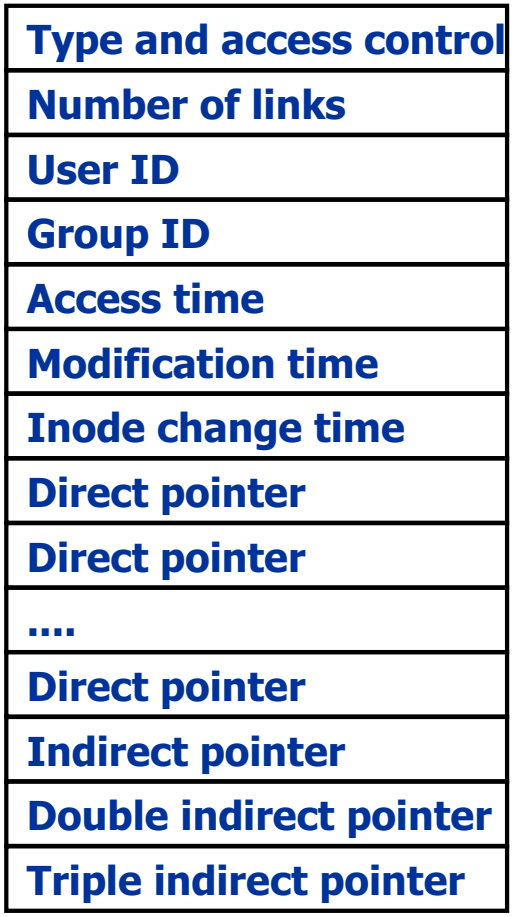
Index blocks called **inodes** (index nodes) in UNIX/Linux

On file open, OS opens **inode table:**

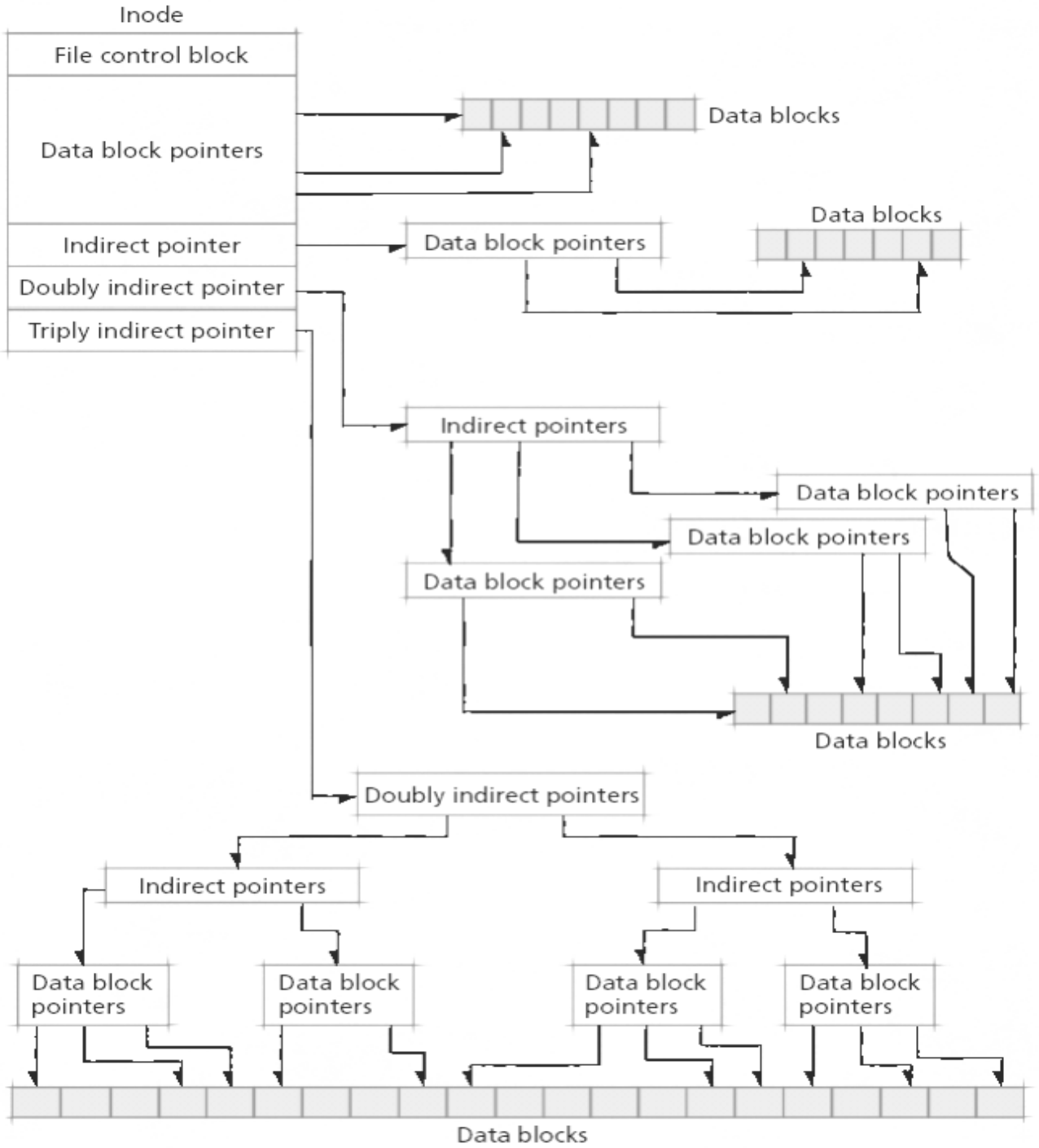
- Inode entry created in memory

Structured as inode on disk, but includes:

1. Disk device number
2. Inode number (for rewrite)
3. Num of processes with opened file
4. Major/minor device number



inode


Inode
File control block
Data block pointers
Indirect pointer
Doubly indirect pointer
Triply indirect pointer
Data blocks
Data block pointers
Data blocks
Indirect pointers
Data block pointers
Data block pointers
Data block pointers
Data blocks
Doubly indirect pointers
Indirect pointers
Indirect pointers
Data block pointers
Data block pointers
Data block pointers
Data block pointers
Data blocks

Example: Inodes I

In a particular OS, an inode contains 6 direct pointers, 1 pointer to a (single) indirect block and 1 pointer to a doubly indirect block.
Each of these pointers is 8 bytes long.
Assume a disk block is 1024 bytes and that each indirect block fills a single block.

1. What is the maximum file size for this file system?

2. What is the maximum file size if the OS would use triply indirect pointers?

Answer: Inodes I

1. The maximum file size is:

   6 x 1024 (data directly indexed)
   + 128 x 1024 (data referenced by single indirect)
   + 128^2 x 1024 (data referenced by double indirect)
   = 16.13 MB

2. The maximum file size is:

   6 x 1024 (data directly indexed)
   + 128 x 1024 (data referenced by single indirect)
   + 128^2 x 1024 (data referenced by double indirect)
   + 128^3 x 1024 (data referenced by triple indirect)
   = 2.02 GB

Example: Inodes II

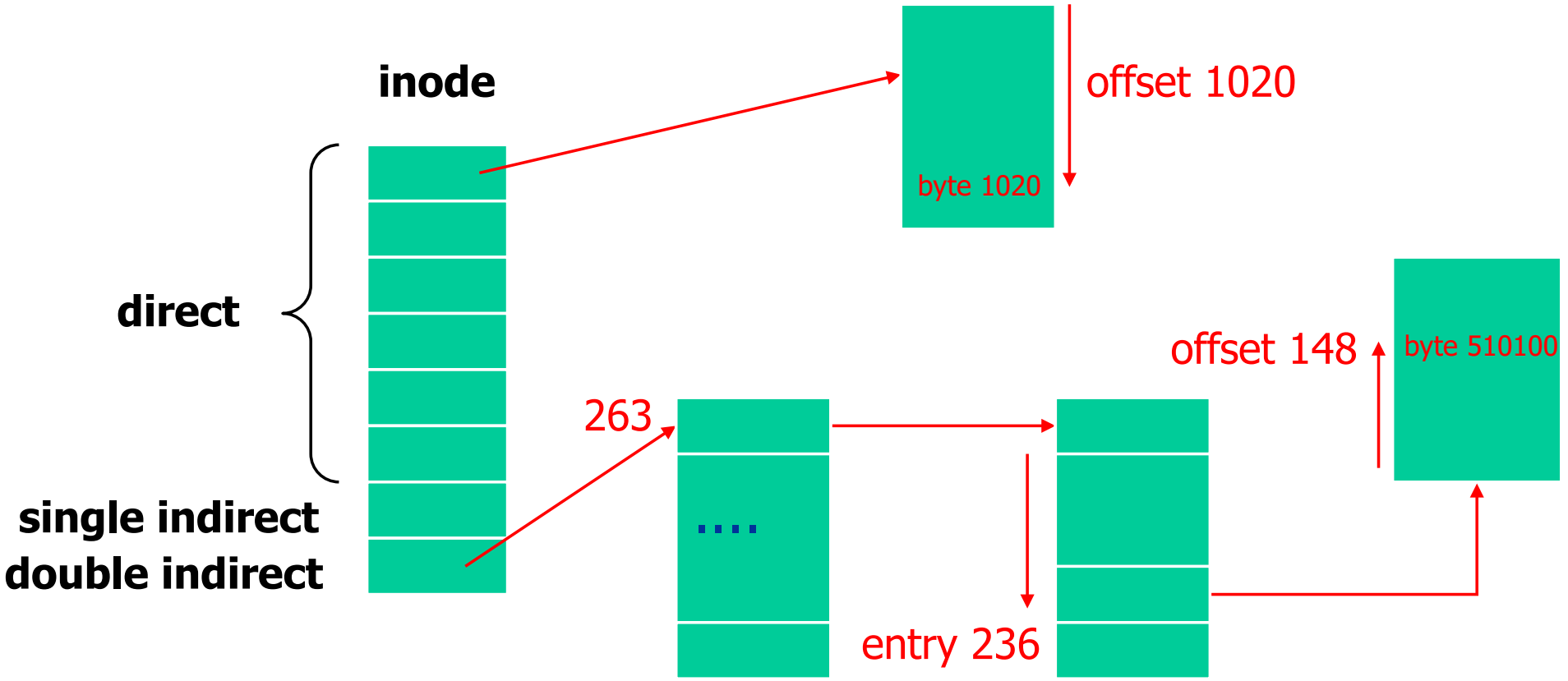
In a particular OS, an inode contains 6 direct pointers, 1 pointer to a (single) indirect block and 1 pointer to a doubly indirect block.

Each of these pointers is 4 bytes long.
Assume a disk block is 1024 bytes and that each indirect block fills a single disk block.

How many disk block reads will be needed to access:

- the 1020^{th} data byte?
- the $510{,}100^{th}$ data byte?

Answer: Inodes II



1020 / 1024 = 1st block

510100 / 1024 = 499th block

Summary: File Allocation Examples

	Block chaining	**File allocation table**	**Inodes**
Byte 1020	2	2	2 (assuming inode not yet in memory)
Byte 510100	501	best case: 3 worst case: 500	4 (assuming inode not yet in memory)

Free Space Management

Problem: Need to manage storage device's free space

- Quick access to free blocks for allocation

Use **free list**

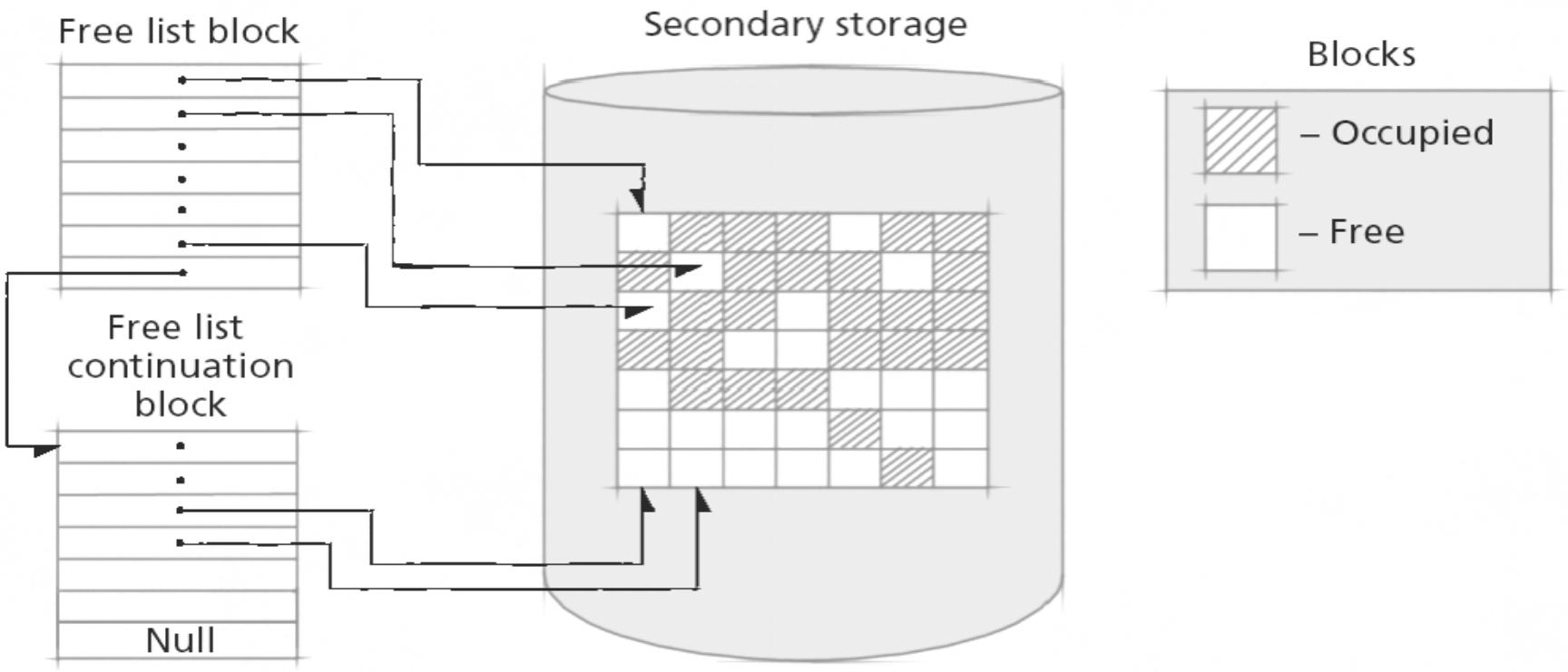
- Linked list of blocks containing locations of free blocks
- Blocks are allocated from beginning of free list
- Newly-freed blocks appended to end of list

Low overhead to perform free list maintenance operations

Files likely to be allocated in noncontiguous blocks

- Increases file access time

Free List


Free list block
Secondary storage
Blocks
– Occupied
– Free
Free list
continuation
block
Null

Bitmap I

Bitmap contains one bit (in memory) for each disk block

- Indicates whether block in use
- i^{th} bit corresponds to i^{th} block on disk

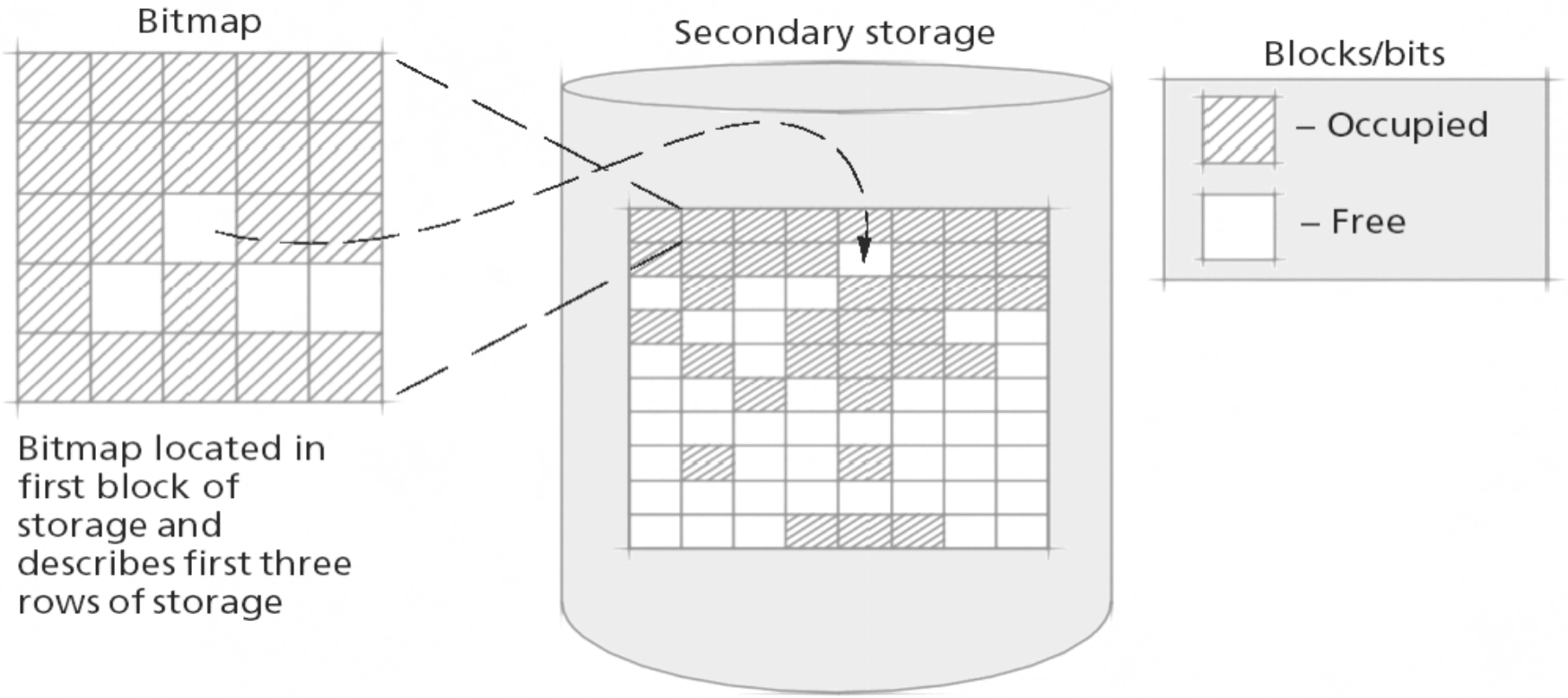
Advantage of bitmaps over free lists

- Can quickly determine available contiguous blocks at certain locations on secondary storage

Disadvantage

- May need to search entire bitmap to find free block, resulting in execution overhead

Bitmap II


Bitmap
Secondary storage
Blocks/bits
– Occupied
– Free
Bitmap located in first block of storage and describes first three rows of storage

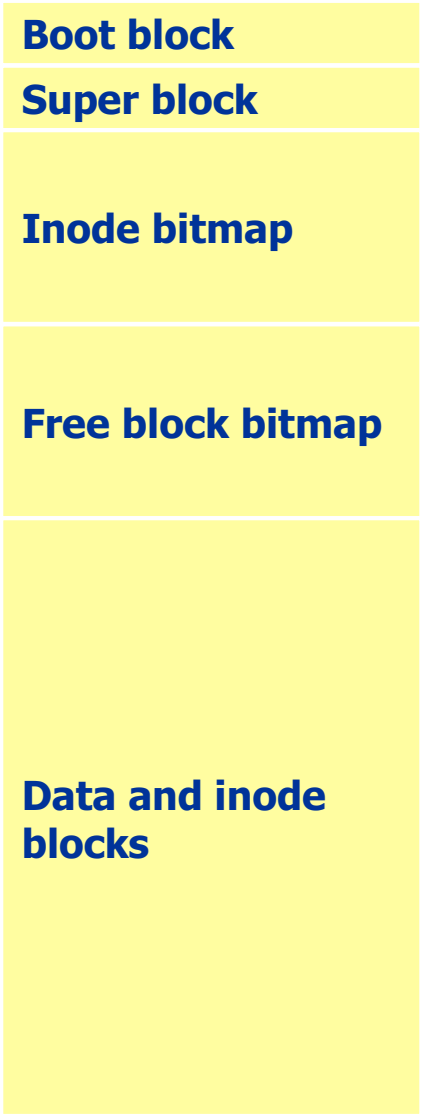
Filesystem Layout

Fixed disk layout (with inodes)

- boot block
- superblock
- free inode bitmap
- free block (zone) bitmap
- inodes + data

Superblock (contains crucial info about FS)

- no of inodes
- no of data blocks
- start of inode & free space bitmap
- first data block
- block size
- maximum file size
- …

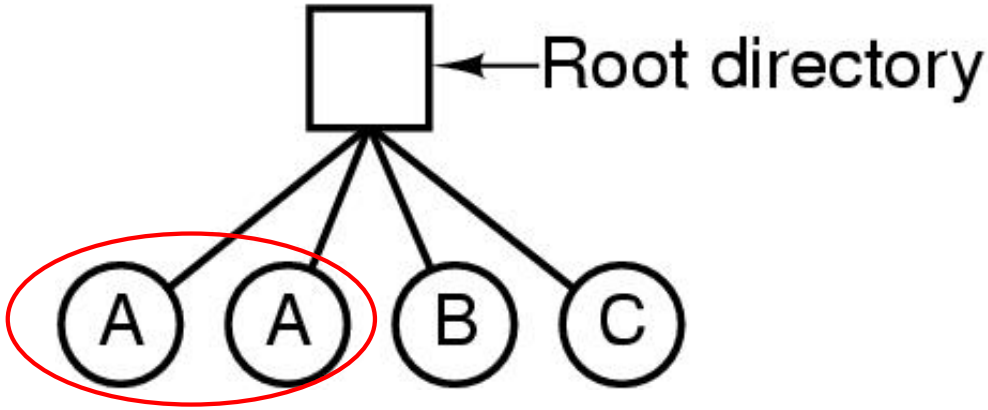


FS Directory Organisation

File System Directories

Directory: Maps symbolic file names to logical disk locations (e. g. `wombat.txt` ➔ disk 0, block 2 (LBA))

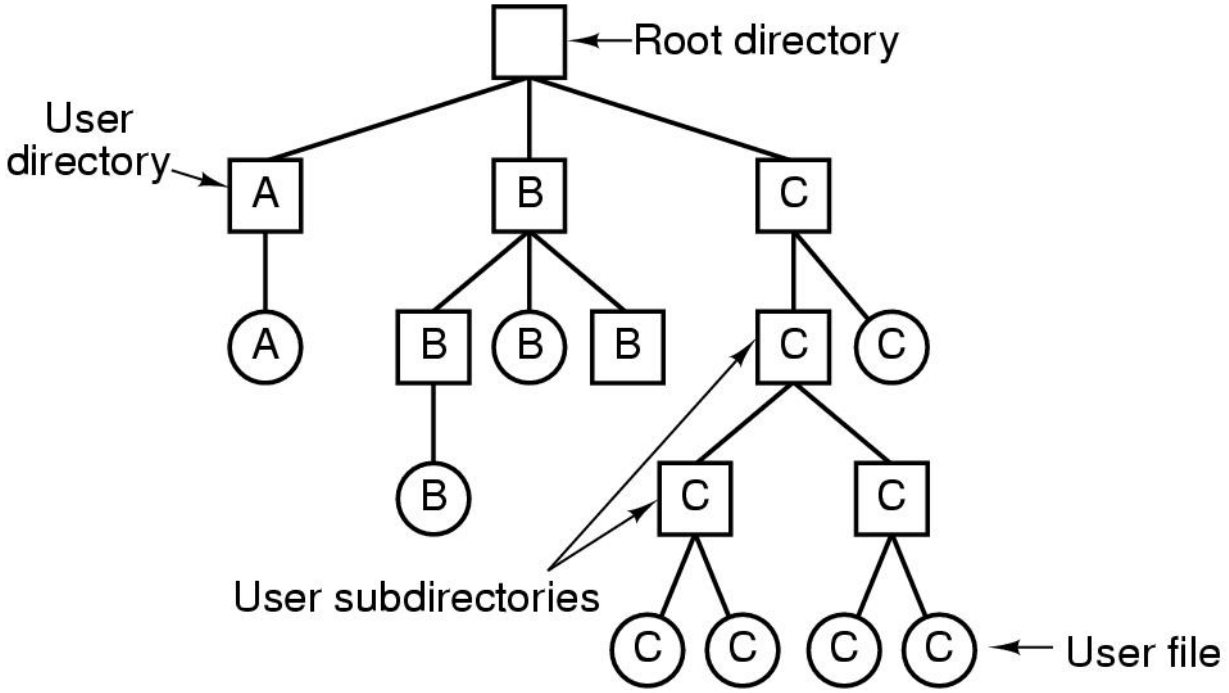
- Helps with file organisation
- Ensures uniqueness of names



MultiLevel (Tree) Directory Structure

Hierarchical file system (e.g. UNIX, Linux, Windows, Mac, ...)

- **Root** indicates where on disk root directory begins
- **Root directory** points to various directories
  - Each of which contains entries for its files
  - File names need be unique only within given directory



Pathnames

Pathname

- File names usually given as path from root directory to file

Absolute pathnames

- Unix/Linux: **`/home/prp/wombat`**
- Windows: **`\home\prp\wombat`**

Relative pathnames

- Relative to working (or current) directory
- Can be changed using **`cd`** command; displayed with **`pwd`**
- Current directory: **`.`** Parent directory: **`..`**

Examples

```
cp /home/prp/wombat /home/prp/wombat.old
cp wombat wombat.old
```

are the same if working directory is **`/home/prp`**

Directory Operations I

Open/close directory

Search:

- Find file in directory system using pattern matching on string, wildcard characters
- e.g. `*.pdf` or `wombat[0-9].c`

Create/delete files or directories

Link: Create link to file Unlink: Remove link to file

Change directory: Opens new directory as current one

Directory Operations II

List: Lists or displays files in directory

- Implemented as multiple read entry operations

Read attributes: Reads attributes of file

Write attributes

- Change attributes of file
- e.g. protection information or name

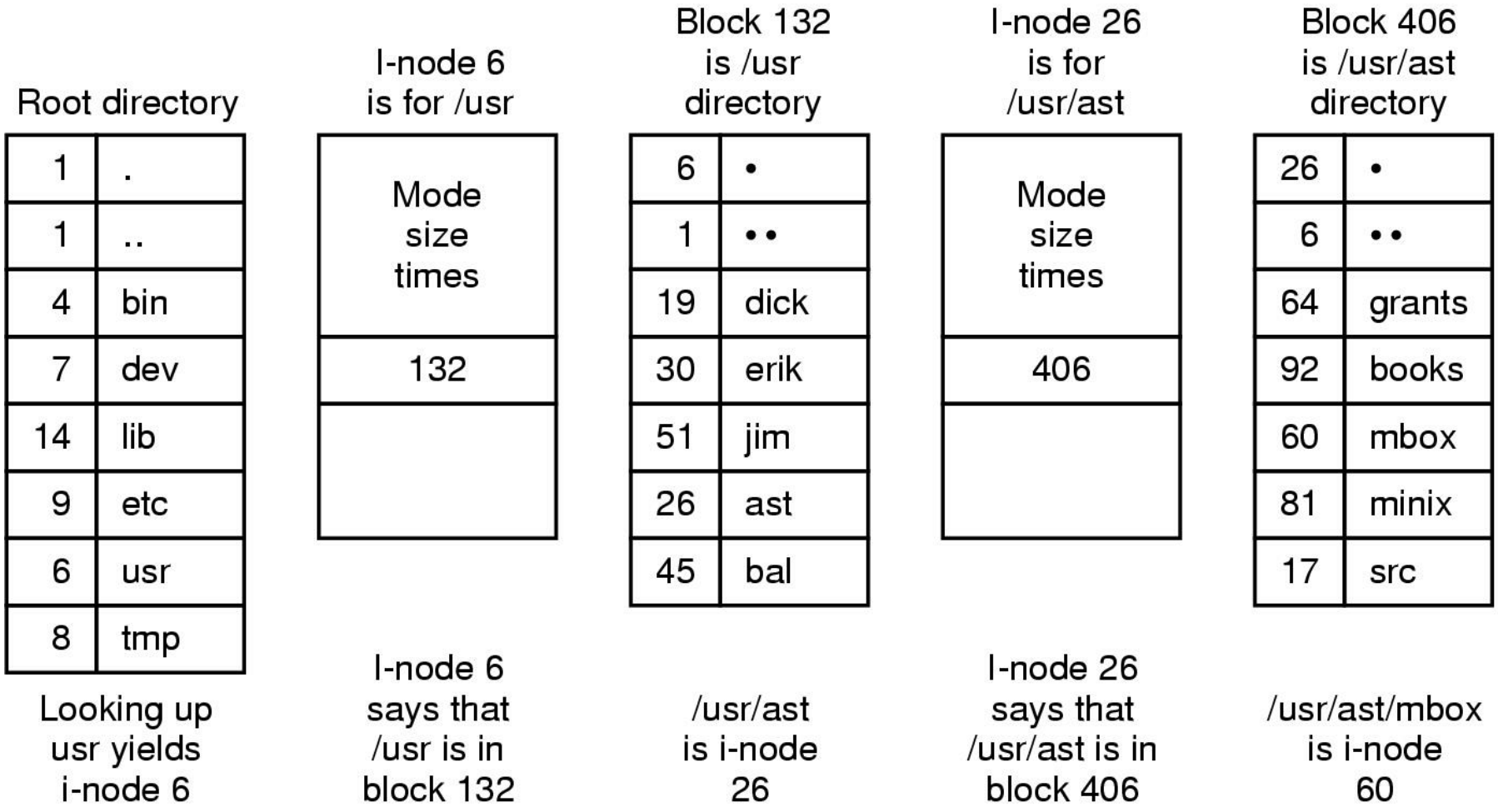
Mount

- Creates link in directory to directory in different file system
- e.g. on another disk or remote server

Unix/Linux: Directory System Calls

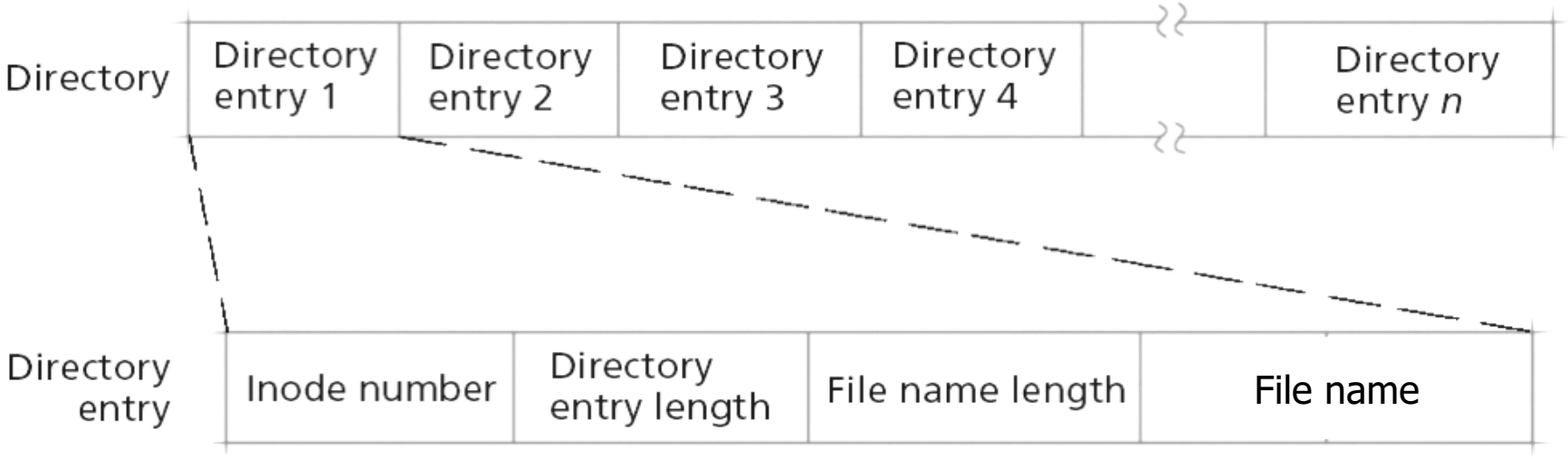
System Call	Description
`s = mkdir (path, mode)`	Create a new directory
`s = rmdir (path)`	Remove directory
`s = link (oldpath, newpath)`	Create a new (hard) link
`s = unlink (path)`	Unlink a file
`s = chdir (path)`	Change working directory
`dir = opendir (path)`	Open directory for reading
`s = closedir (dir)`	Close directory
`dirent = readdir (dir)`	Read one entry from directory
`rewinddir (dir)`	Rewind directory to re-read

Unix/Linux: Looking Up File Names



Steps in looking up `/usr/ast/mbox`

Linux: Directory Representation




```
struct dirent
{
  long d_ino;                    /* inode number */
  off_t d_off;                   /* offset to this dirent */
  unsigned short d_reclen;       /* length of this d_name */
  char d_name [NAME_MAX+1];      /* file name (null-
                                    terminated) */
}
```

Links

Link: Reference to directory/file in another part of FS

- Allows alternative names (and different locations in tree)

Hard link: Reference address of file

- Only supported for files in Unix (why?)

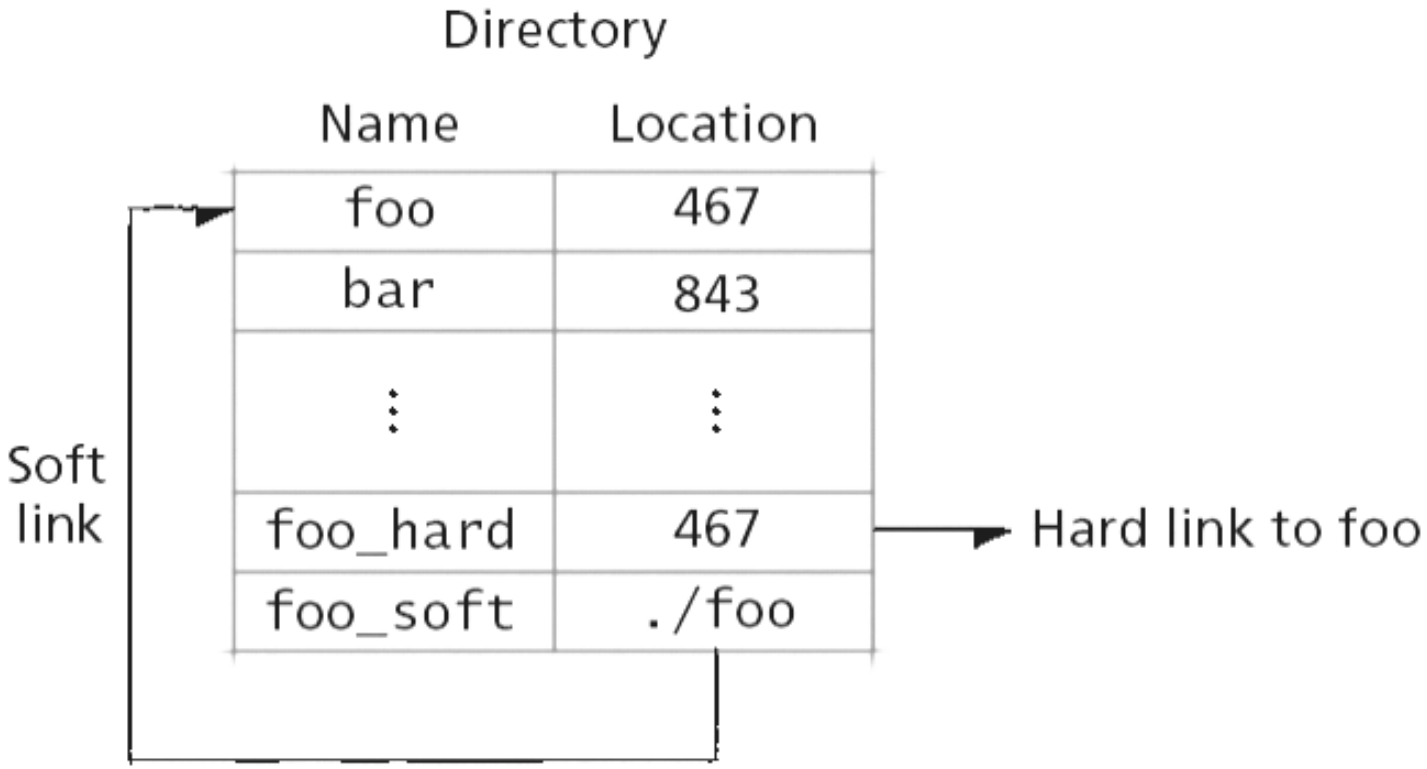
Symbolic (soft) link: Reference full pathname of file/dir

- Created as directory entry

Problems

- File deletion: search for links and remove them
  - Leave links and cause exception when used (symbolic links)
  - Keep link count with file; delete file when count=0 (hard links)
- Looping: directory traversal algorithms may loop

Hard Links vs. Soft Links



Mounting

Mount operation

- Combines multiple FSs into one namespace
- Allows reference from single root directory
- Support for soft-links to files in mounted FSs
  - But not hard-links (why?)

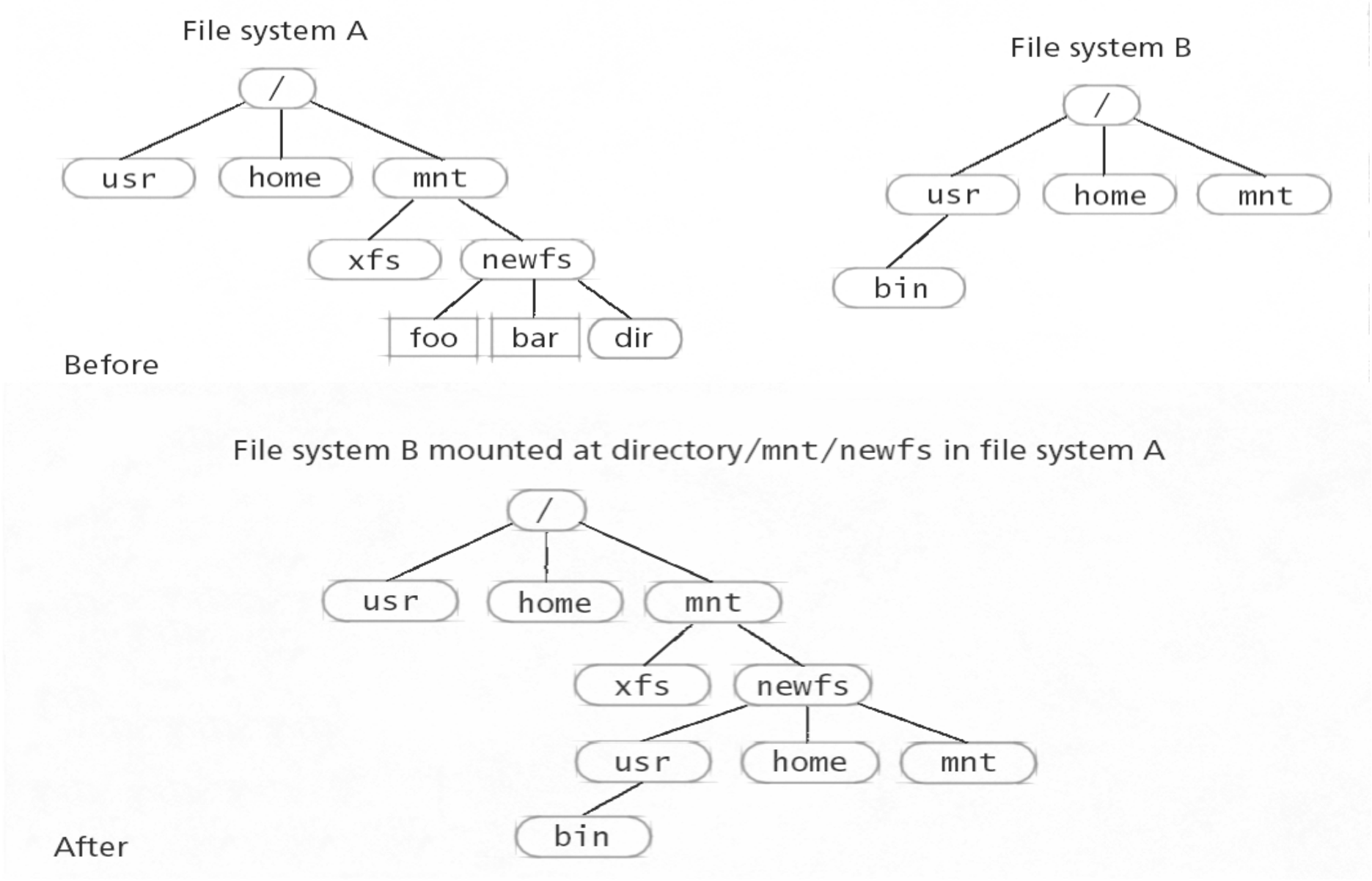
Mount point

- Directory in native FS assigned to root of mounted FS

FSs manage mounted directories with **mount tables**

- Information about location of mount points and devices
- When native FS encounters mount point, use mount table to determine device and type of mounted FS

Example: Mounting



Linux ext2fs

Linux: ext2fs

Goal: high-performance, robust FS with support for advanced features

Typical block sizes: 1024, 2048, 4096 or 8192 bytes

Safety mechanism: 5% of blocks reserved for root

- Allow root processes to continue to run after malicious/errant user process consumes all FS disk space

Linux: ext2fs Inode

ext2 inode

- Represents files and directories in ext2 FS
- Stores information relevant to single file/directory
  - e.g. time stamps, permissions, owner, pointers to data blocks

ext2 inode pointers

- First 12 pointers directly locate 12 data blocks
- 13th pointer is **indirect pointer**
  - Locates block of pointers to data blocks
- 14th pointer is a **doubly-indirect pointer**
  - Locates block of indirect pointers
- 15th pointer is **triply-indirect pointer**
  - Locates block of doubly indirect pointers

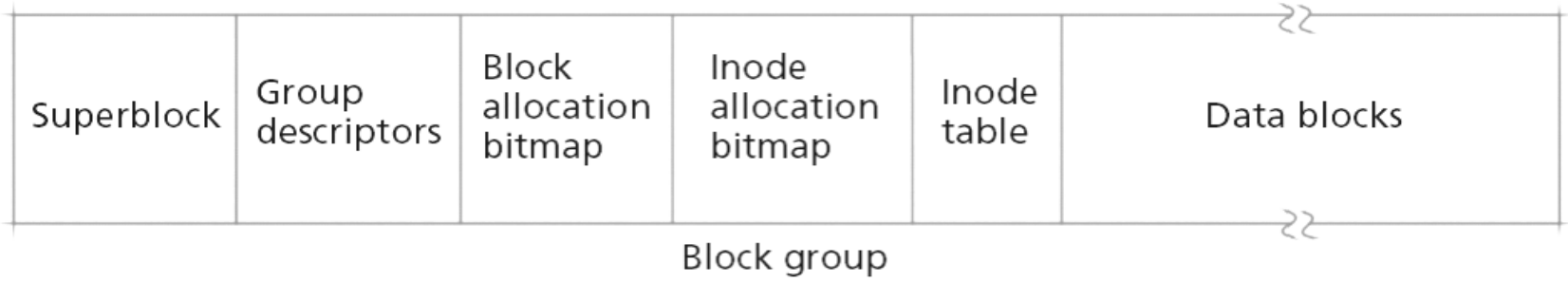
Provides fast access to small files, while supporting very large files

Linux: ext2fs Block Groups I

Block groups

- Clusters of contiguous blocks
- FS attempts to store related data in same block group
- Reduces seek time for accessing groups of related data

Block group structure



- **Superblock:** Critical data about entire FS
  - e.g. total num of blocks and inodes, size of block groups, time FS was mounted, ...
  - Redundant copies of superblock in some block groups

Linux: ext2fs Block Groups II

- **Inode table:** Contains entry for each inode in block group
- **Inode allocation bitmap:** Inodes used within block group
- **Block allocation bitmaps:** Blocks used within group
- **Group descriptor:** Block numbers for location of:
  - inode allocation bitmap
  - block allocation bitmap
  - inode table
  - accounting information
- **Data blocks:** Remaining blocks store file/directory data
  - Directory information stored in directory entries
  - Each directory entry is composed of: inode number, directory entry length, file name length, file type, file name

Linux: ext2fs Block Groups III