Chapter 6: File Systems

File systems

- Files
- Directories & naming
- File system implementation
- Example file systems

Long-term information storage

- Must store large amounts of data
  - Gigabytes -> terabytes -> petabytes
- Stored information must survive the termination of the process using it
  - Lifetime can be seconds to years
  - Must have some way of finding it!
- Multiple processes must be able to access the information concurrently

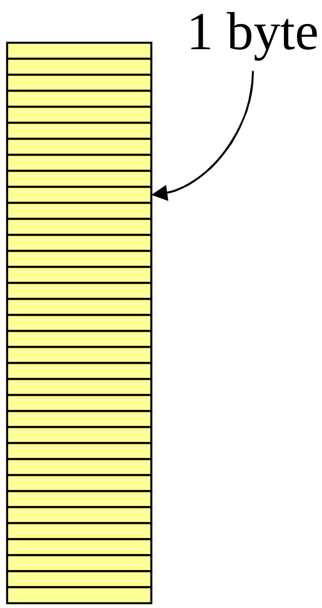
Naming files

- Important to be able to *find* files after they're created
- Every file has at least one name
- Name can be
  - Human-accessible: "foo.c", "my photo", "Go Slugs!"
  - Machine-usable: 4502, 33481
- Case may or may not matter
  - Depends on the file system
- Name may include information about the file's contents
  - Certainly does for the user (the name should make it easy to figure out what's in it!)
  - Computer may use part of the name to determine the file type

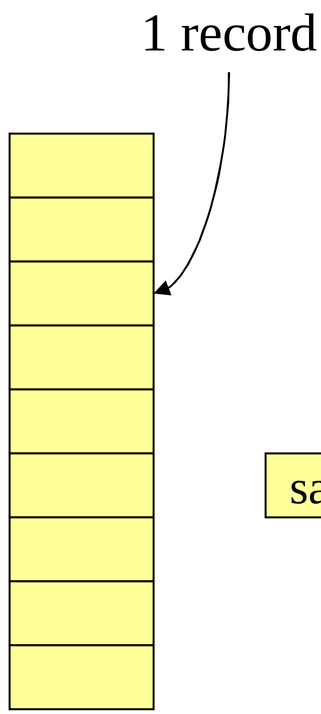
Typical file extensions

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

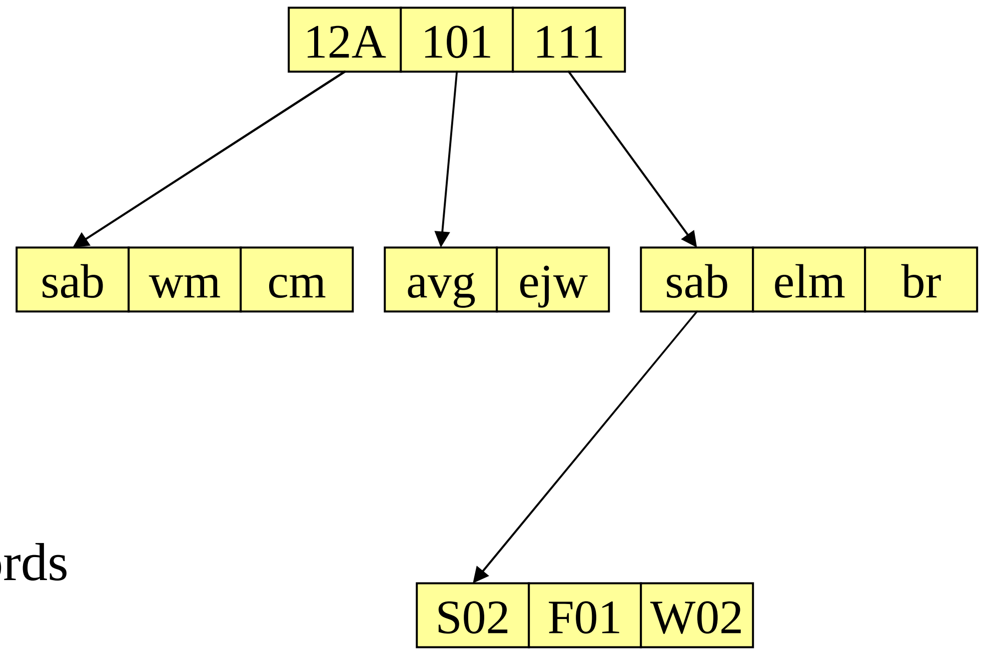
File structures



Sequence of bytes



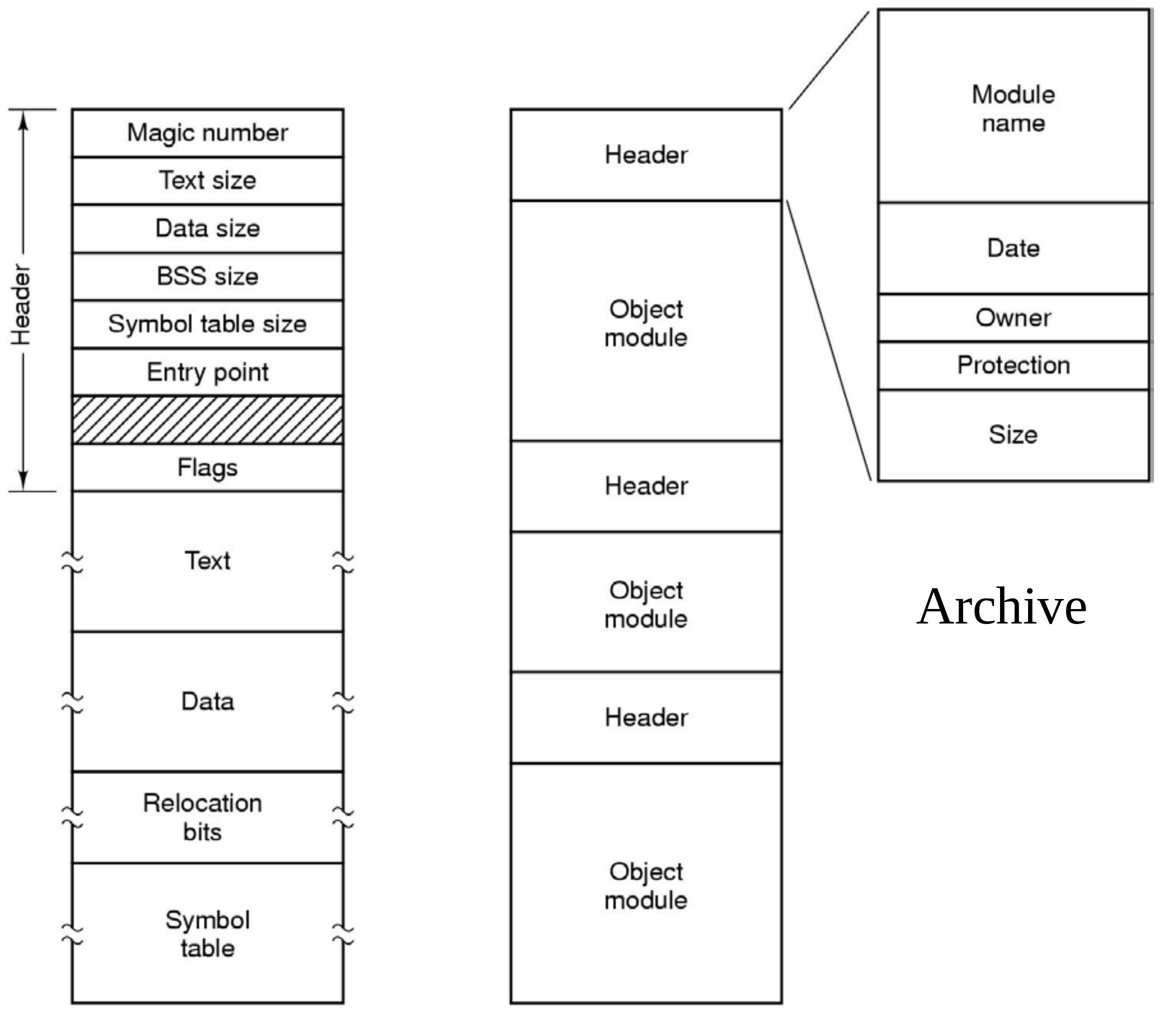
Sequence of records



Tree

File types

Executable file



(a) (b)

Archive

Accessing a file

- Sequential access
  - Read all bytes/records from the beginning
  - Cannot jump around
    - May rewind or back up, however
  - Convenient when medium was magnetic tape
  - Often useful when whole file is needed
- Random access
  - Bytes (or records) read in any order
  - Essential for database systems
  - Read can be …
    - Move file marker (seek), then read or …
    - Read and then move file marker

File attributes

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

File operations

- Create: make a new file
- Delete: remove an existing file
- Open: prepare a file to be accessed
- Close: indicate that a file is no longer being accessed
- Read: get data from a file
- Write: put data to a file
- Append: like write, but only at the end of the file
- Seek: move the “current” pointer elsewhere in the file
- Get attributes: retrieve attribute information
- Set attributes: modify attribute information
- Rename: change a file’s name

Using file system calls

```
/* 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 */
```

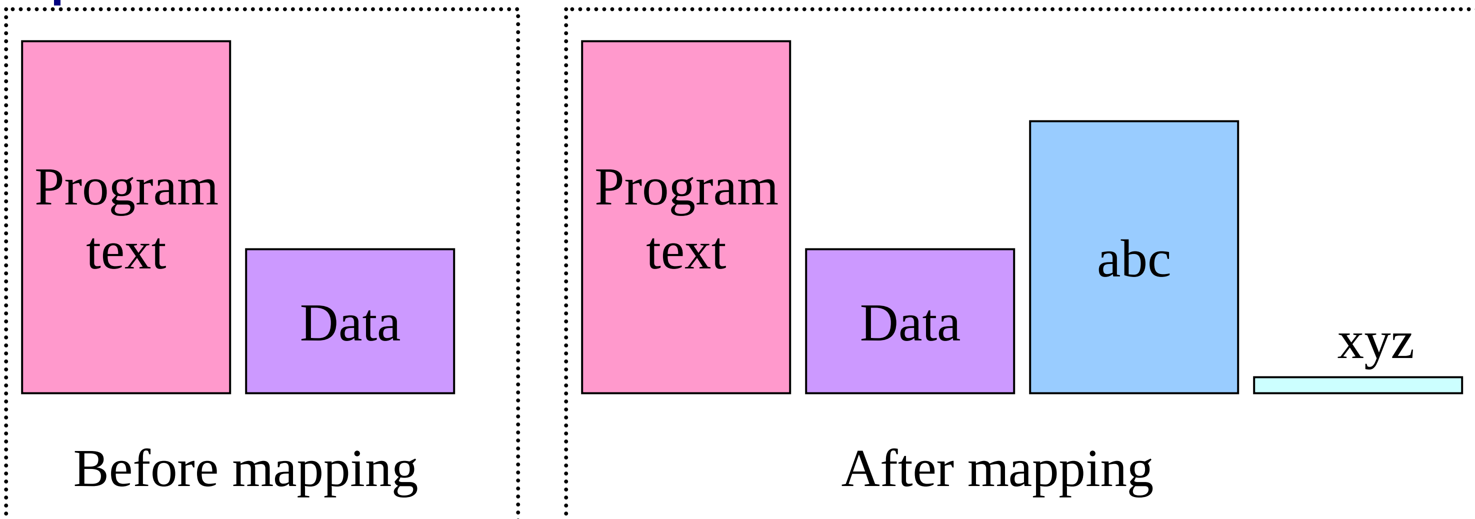
Using file system calls, continued

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

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

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

Memory-mapped files



- Segmented process before mapping files into its address space
- Process after mapping
  - Existing file *abc* into one segment
  - Creating new segment for *xyz*

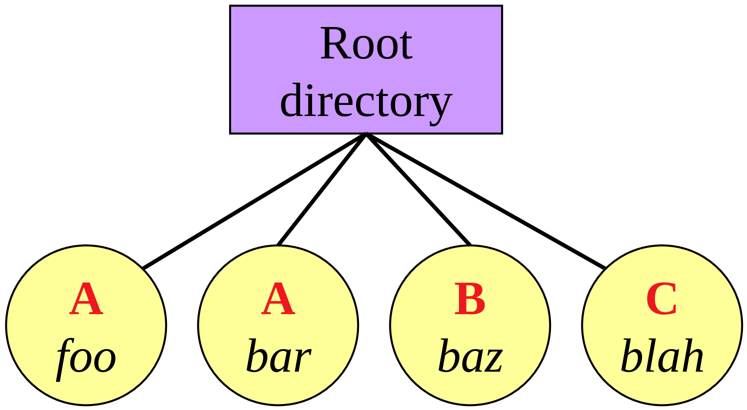
More on memory-mapped files

- Memory-mapped files are a convenient abstraction
  - Example: string search in a large file can be done just as with memory!
  - Let the OS do the buffering (reads & writes) in the virtual memory system
- Some issues come up…
  - How long is the file?
    - Easy if read-only
    - Difficult if writes allowed: what if a write is past the end of file?
  - What happens if the file is shared: when do changes appear to other processes?
  - When are writes flushed out to disk?
- Clearly, easier to memory map read-only files…

Directories

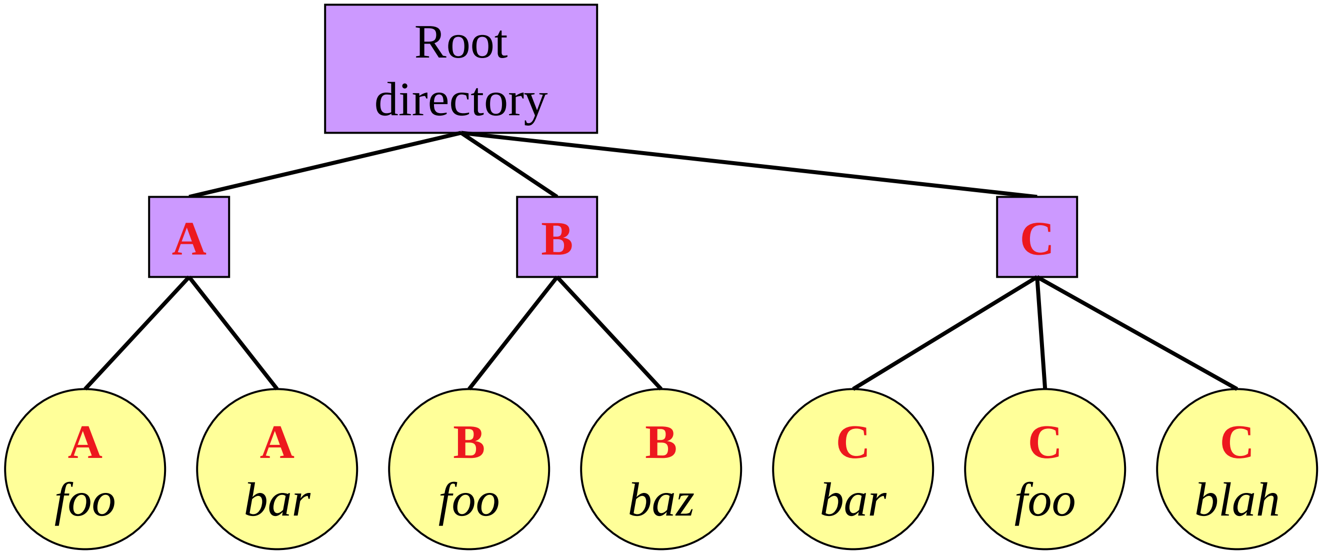
- Naming is nice, but limited
- Humans like to group things together for convenience
- File systems allow this to be done with *directories* (sometimes called *folders*)
- Grouping makes it easier to
  - Find files in the first place: remember the enclosing directories for the file
  - Locate related files (or just determine which files are related)

Single-level directory systems



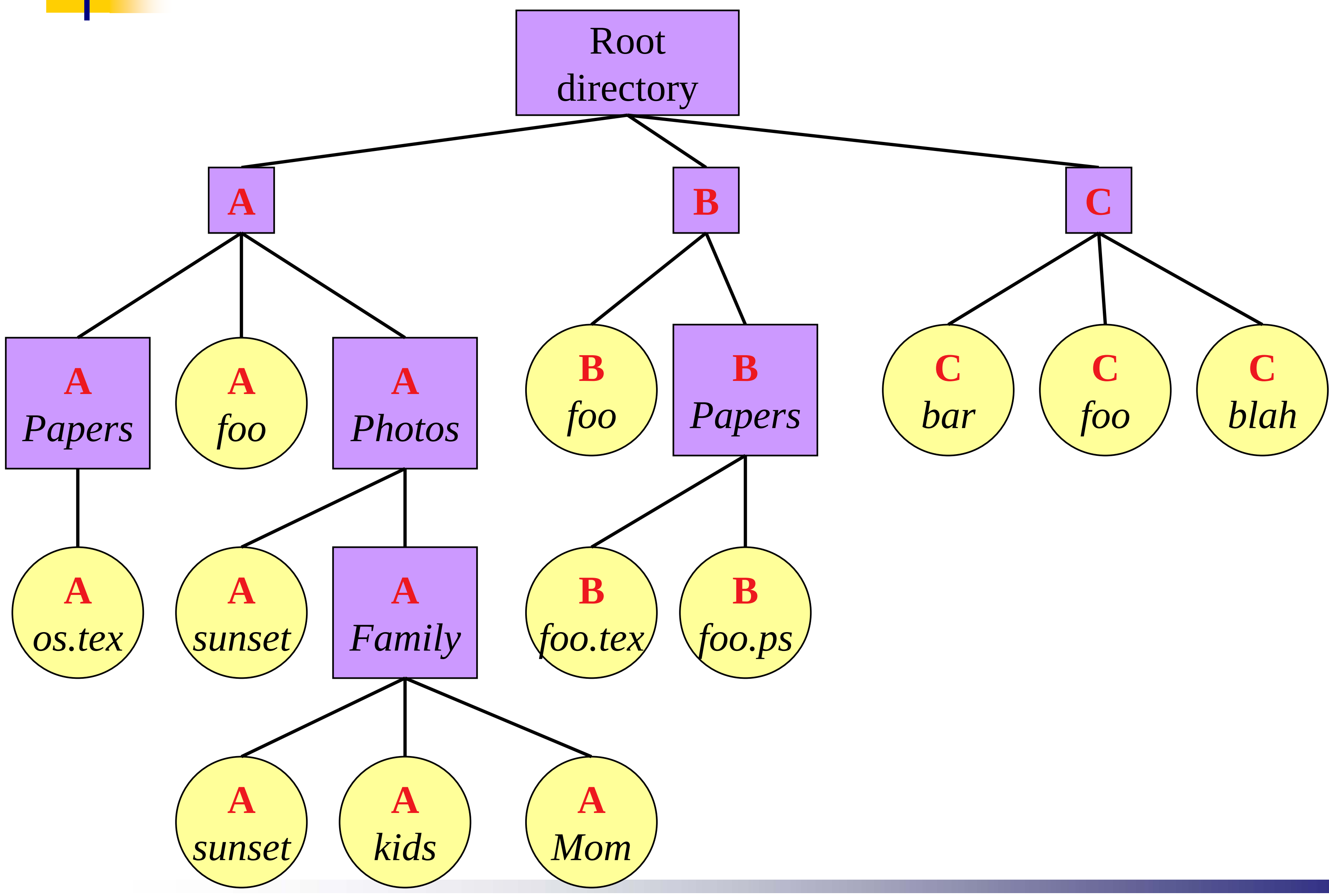
- One directory in the file system
- Example directory
  - Contains 4 files (*foo*, *bar*, *baz*, *blah*)
  - owned by 3 different people: A, B, and C (owners shown in red)
- Problem: what if user B wants to create a file called *foo*?

Two-level directory system

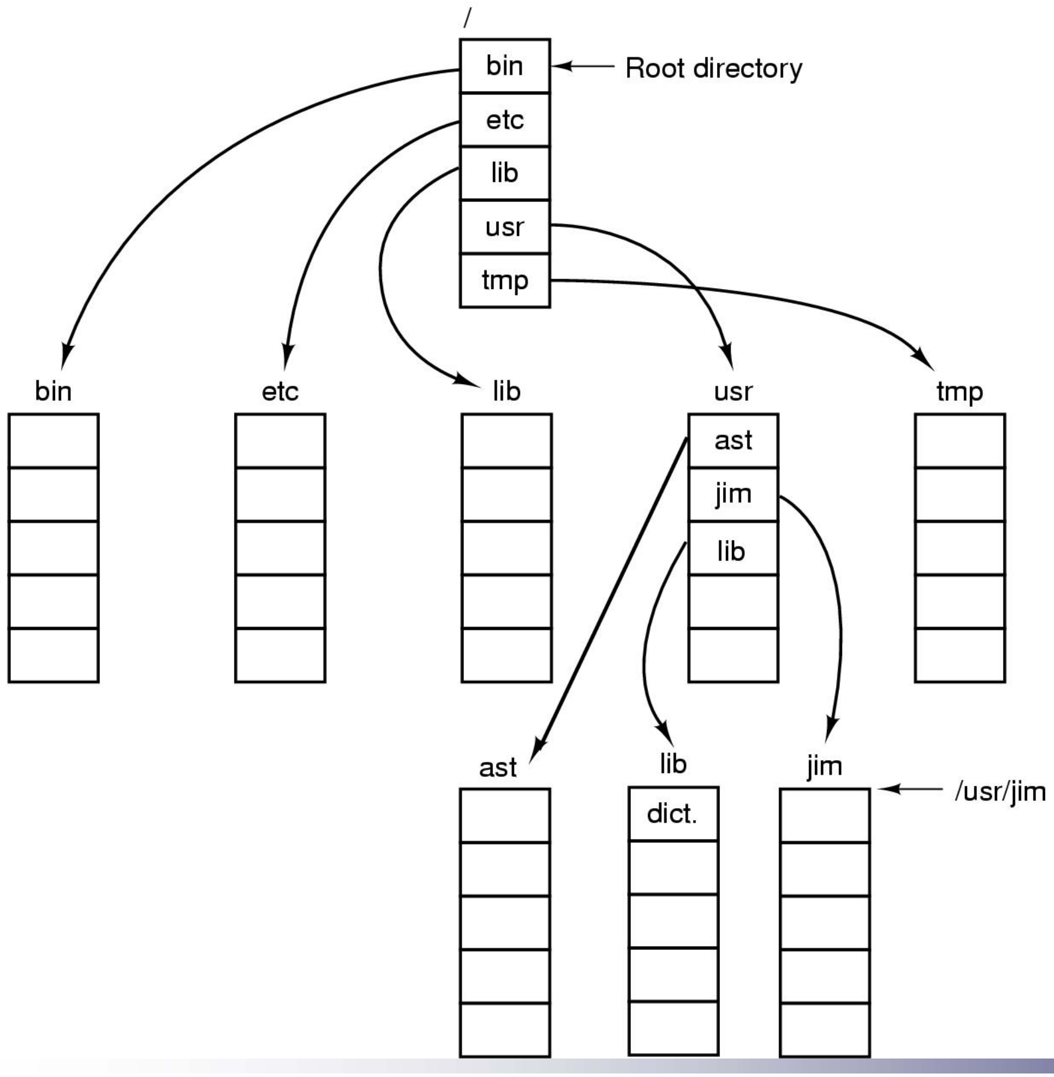


- Solves naming problem: each user has her own directory
- Multiple users can use the same file name
- By default, users access files in their own directories
- Extension: allow users to access files in others' directories

Hierarchical directory system



Unix directory tree



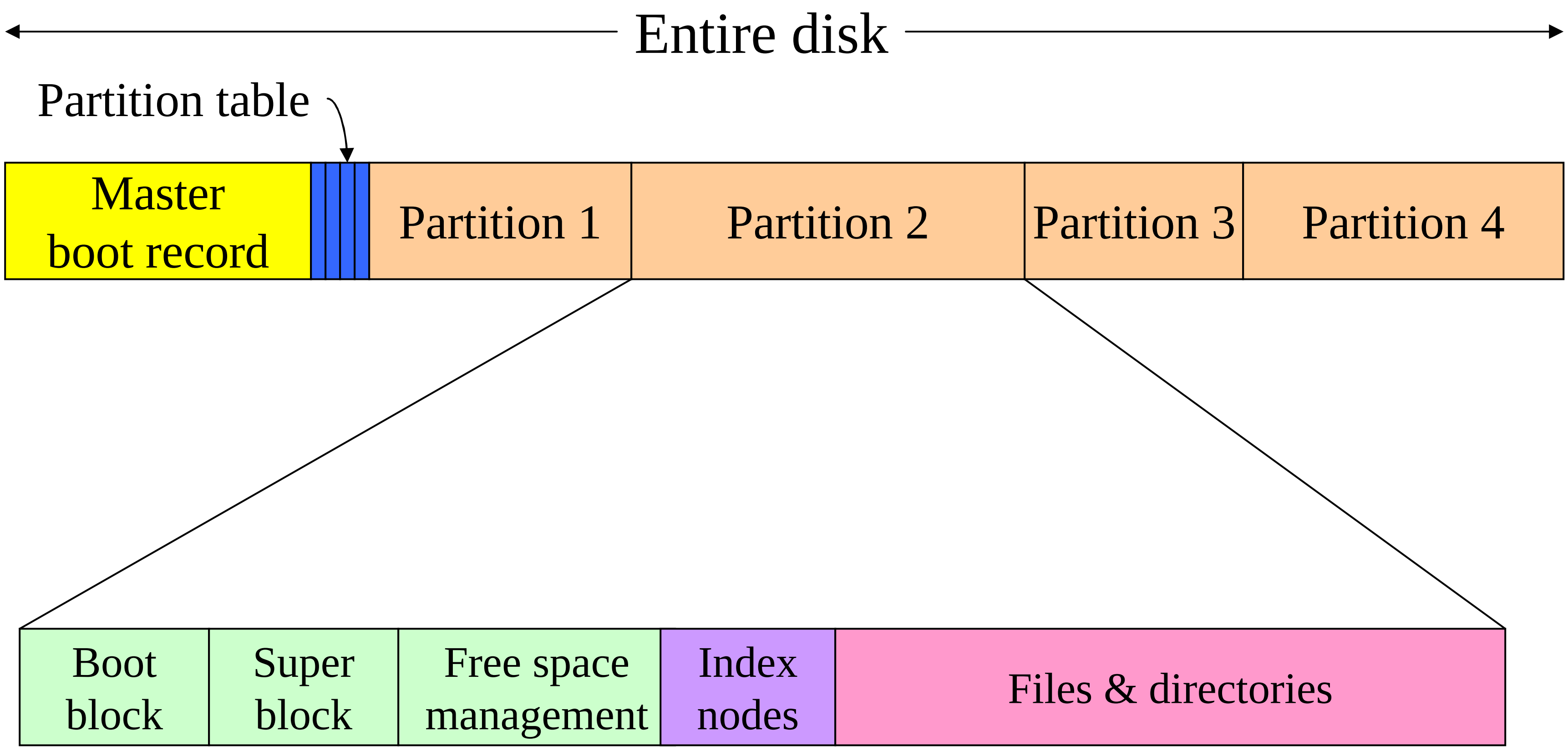
Operations on directories

- Create: make a new directory
- Delete: remove a directory (usually must be empty)
- Opendir: open a directory to allow searching it
- Closedir: close a directory (done searching)
- Readdir: read a directory entry
- Rename: change the name of a directory
  - Similar to renaming a file
- Link: create a new entry in a directory to link to an existing file
- Unlink: remove an entry in a directory
  - Remove the file if this is the last link to this file

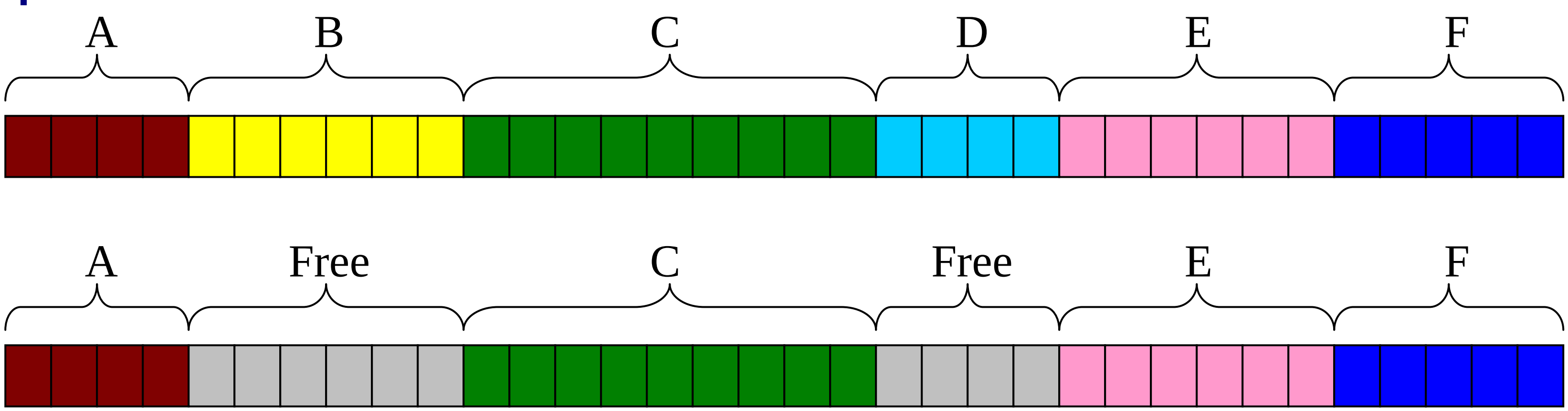
File system implementation issues

- How are disks divided up into file systems?
- How does the file system allocate blocks to files?
- How does the file system manage free space?
- How are directories handled?
- How can the file system improve…
  - Performance?
  - Reliability?

Carving up the disk



Contiguous allocation for file blocks

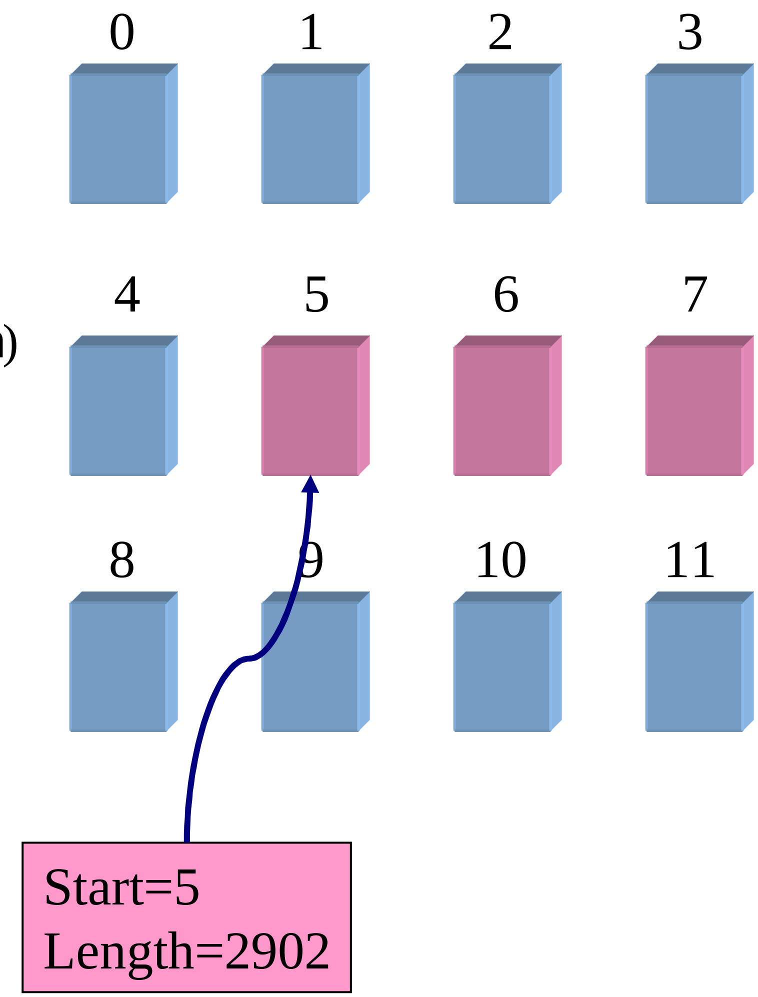


- Contiguous allocation requires all blocks of a file to be consecutive on disk
- Problem: deleting files leaves “holes”
  - Similar to memory allocation issues
  - Compacting the disk can be a very slow procedure…

Contiguous allocation

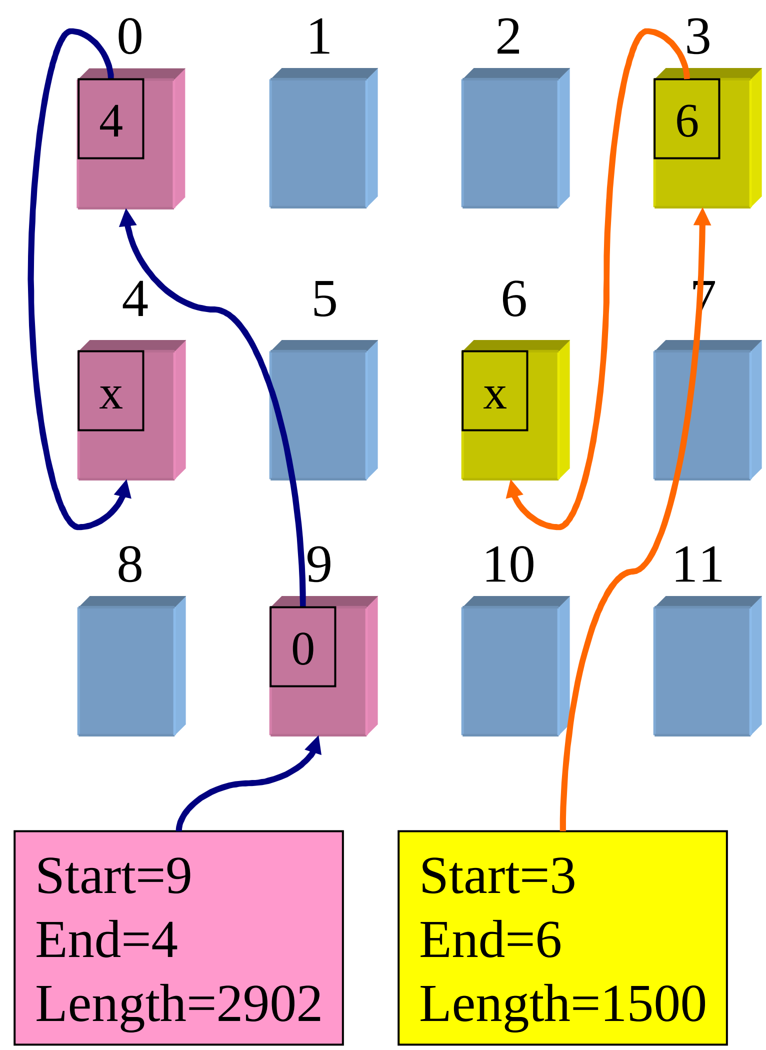
- Data in each file is stored in consecutive blocks on disk
- Simple & efficient indexing
  - Starting location (block #) on disk (`start`)
  - Length of the file in blocks (`length`)
- Random access well-supported
- Difficult to grow files
  - Must pre-allocate all needed space
  - Wasteful of storage if file isn't using all of the space
- Logical to physical mapping is easy

```
blocknum = (pos / 1024) + start;
offset_in_block = pos % 1024;
```



Linked allocation

- File is a linked list of disk blocks
  - Blocks may be scattered around the disk drive
  - Block contains both pointer to next block and data
  - Files may be as long as needed
- New blocks are allocated as needed
  - Linked into list of blocks in file
  - Removed from list (bitmap) of free blocks



Finding blocks with linked allocation

- Directory structure is simple
  - Starting address looked up from directory
  - Directory only keeps track of first block (not others)
- No wasted space - all blocks can be used
- Random access is difficult: must always start at first block!
- Logical to physical mapping is done by

```
block = start;
offset_in_block = pos % 1020;
for (j = 0; j < pos / 1020; j++) {
  block = block->next;
}
```

  - Assumes that *next* pointer is stored at end of block
  - May require a long time for seek to random location in file

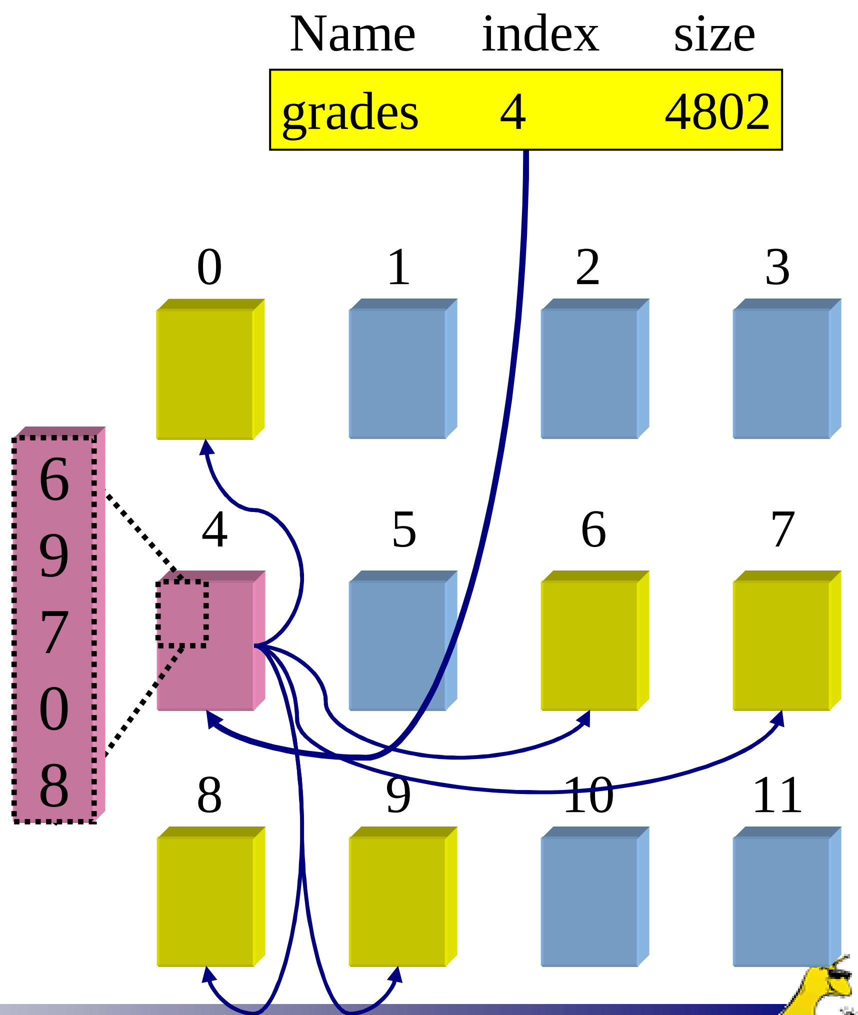
Linked allocation using a RAM-based table

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

- Links on disk are slow
- Keep linked list in memory
- Advantage: faster
- Disadvantages
  - Have to copy it to disk at some point
  - Have to keep in-memory and on-disk copy consistent

Using a block index for allocation

- Store file block addresses in an array
  - Array itself is stored in a disk block
  - Directory has a pointer to this disk block
  - Non-existent blocks indicated by -1
- Random access easy
- Limit on file size?



Finding blocks with indexed allocation

- Need location of index table: look up in directory
- Random & sequential access both well-supported: look up block number in index table
- Space utilization is good
  - No wasted disk blocks (allocate individually)
  - Files can grow and shrink easily
  - Overhead of a single disk block per file
- Logical to physical mapping is done by
  block = index[block % 1024];
  offset_in_block = pos % 1024;
- Limited file size: 256 pointers per index block, 1 KB per file block -> 256 KB per file limit

Larger files with indexed allocation

- How can indexed allocation allow files larger than a single index block?
- Linked index blocks: similar to linked file blocks, but using index blocks instead
- Logical to physical mapping is done by

```
index = start;
blocknum = pos / 1024;
for (j = 0; j< blocknum /255); j++) {
 index = index->next;
}
block = index[blocknum % 255];
offset_in_block = pos % 1024;
```

- File size is now unlimited
- Random access slow, but only for very large files

Two-level indexed allocation

- Allow larger files by creating an index of index blocks
  - File size still limited, but much larger
  - Limit for 1 KB blocks = 1 KB * 256 * 256 = 226 bytes = 64 MB
- Logical to physical mapping is done by

```
blocknum = pos / 1024;
index = start[blocknum / 256)];
block = index[blocknum % 256]
offset_in_block = pos % 1024;
```

  - Start is the only pointer kept in the directory
  - Overhead is now at least two blocks per file
- This can be extended to more than two levels if larger files are needed...

Block allocation with extents

- Reduce space consumed by index pointers
  - Often, consecutive blocks in file are sequential on disk
  - Store <block,count> instead of just <block> in index
  - At each level, keep total count for the index for efficiency
- Lookup procedure is:
  - Find correct index block by checking the starting file offset for each index block
  - Find correct <block,count> entry by running through index block, keeping track of how far into file the entry is
  - Find correct block in <block,count> pair
- More efficient if file blocks tend to be consecutive on disk
  - Allocating blocks like this allows faster reads & writes
  - Lookup is somewhat more complex

Managing free space: bit vector

- Keep a bit vector, with one entry per file block
  - Number bits from 0 through *n*-1, where *n* is the number of file blocks on the disk
  - If bit[j] == 0, block *j* is free
  - If bit[j] == 1, block *j* is in use by a file (for data or index)
- If words are 32 bits long, calculate appropriate bit by:

```
wordnum = block / 32;
bitnum = block % 32;
```

- Search for free blocks by looking for words with bits unset (words != 0xffffffff)
- Easy to find consecutive blocks for a single file
- Bit map must be stored on disk, and consumes space
  - Assume 4 KB blocks, 8 GB disk => 2M blocks
  - 2M bits = 2^{21} bits = 2^{18} bytes = 256KB overhead

Managing free space: linked list

- Use a linked list to manage free blocks
  - Similar to linked list for file allocation
  - No wasted space for bitmap
  - No need for random access unless we want to find consecutive blocks for a single file
- Difficult to know how many blocks are free unless it's tracked elsewhere in the file system
- Difficult to group nearby blocks together if they're freed at different times
  - Less efficient allocation of blocks to files
  - Files read & written more because consecutive blocks not nearby

Issues with free space management

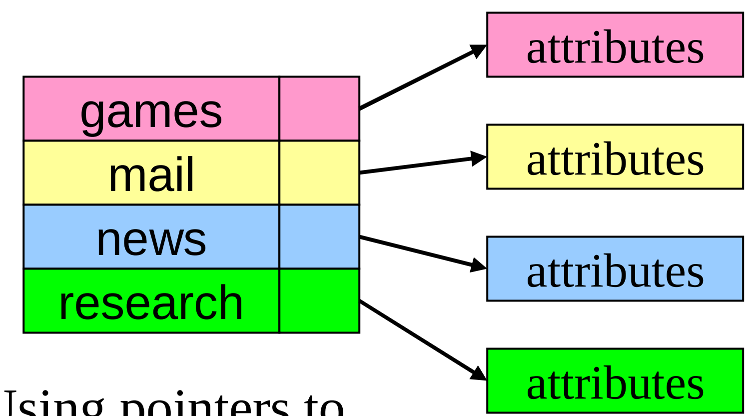
- OS must protect data structures used for free space management
- OS must keep in-memory and on-disk structures consistent
  - Update free list when block is removed: change a pointer in the previous block in the free list
  - Update bit map when block is allocated
    - Caution: on-disk map must ***never*** indicate that a block is free when it's part of a file
    - Solution: set bit[j] in free map to 1 on disk before using block[j] in a file and setting bit[j] to 1 in memory
    - New problem: OS crash may leave bit[j] == 1 when block isn't actually used in a file
    - New solution: OS checks the file system when it boots up…
- Managing free space is a big source of slowdown in file systems

What's in a directory?

- Two types of information
  - File names
  - File metadata (size, timestamps, etc.)
- Basic choices for directory information
  - Store all information in directory
    - Fixed size entries
    - Disk addresses and attributes in directory entry
  - Store names & pointers to index nodes (i-nodes)

games	attributes
mail	attributes
news	attributes
research	attributes

Storing all information in the directory

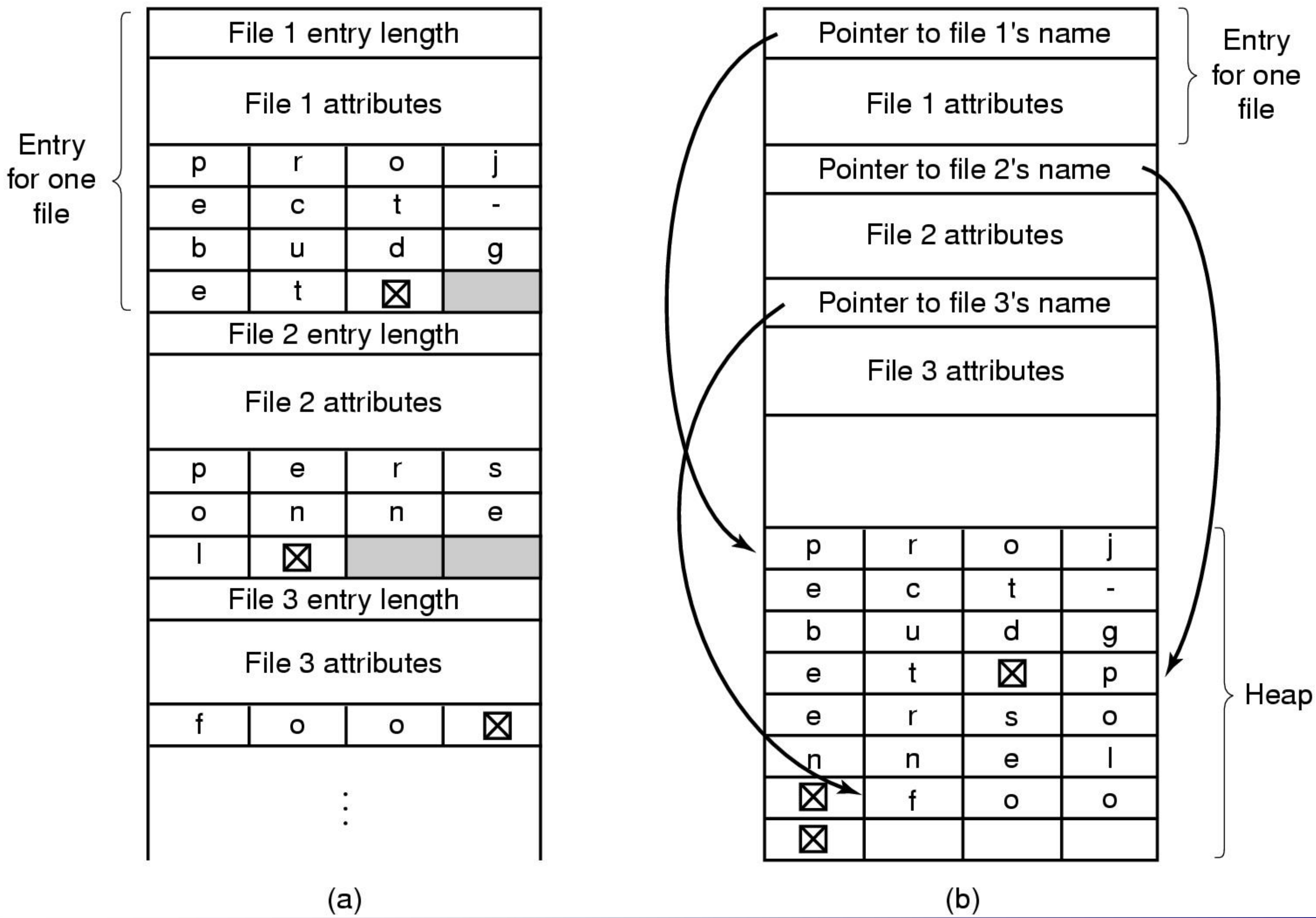


Using pointers to index nodes

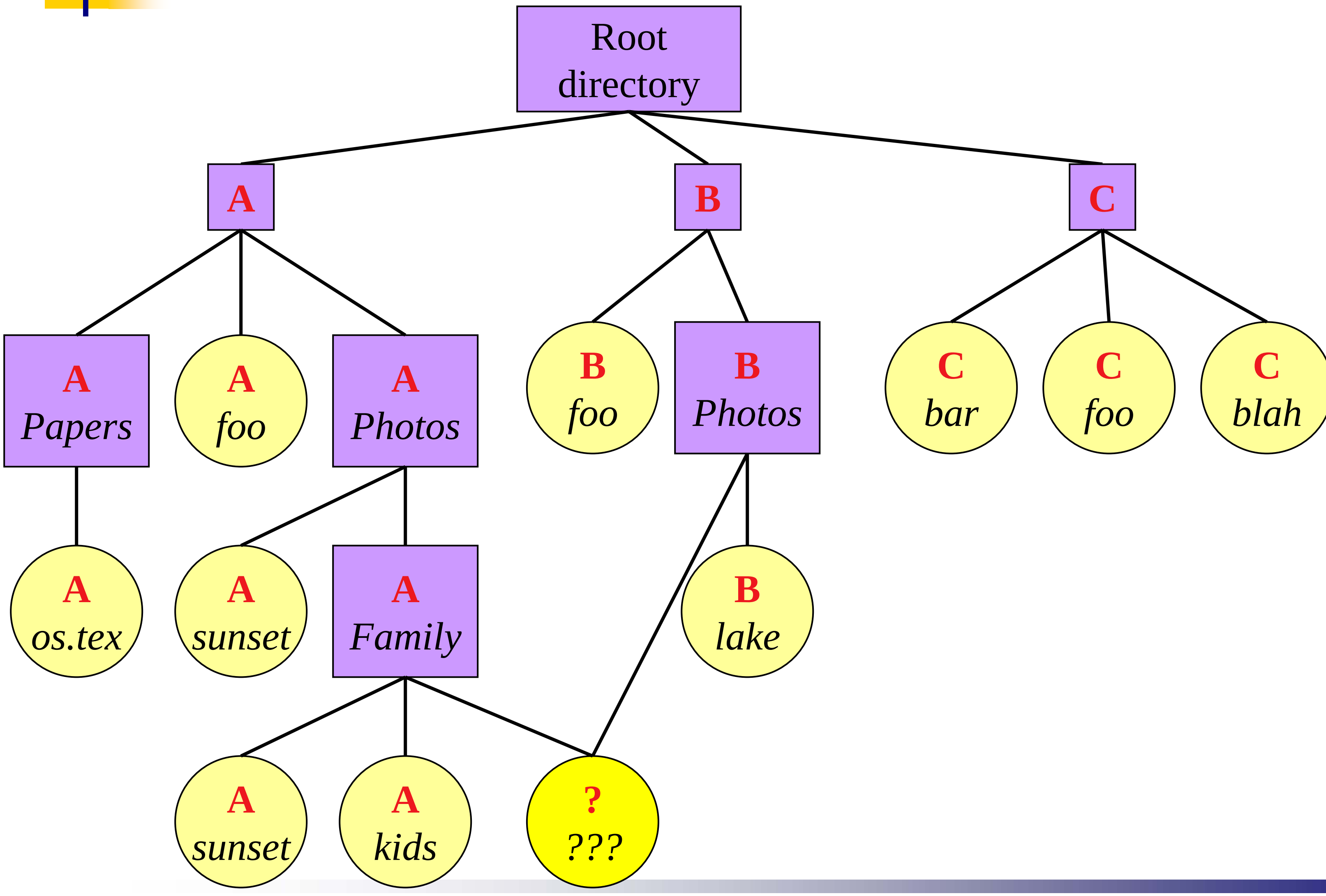
Directory structure

- Structure
  - Linear list of files (often itself stored in a file)
    - Simple to program
    - Slow to run
    - Increase speed by keeping it sorted (insertions are slower!)
  - Hash table: name hashed and looked up in file
    - Decreases search time: no linear searches!
    - May be difficult to expand
    - Can result in collisions (two files hash to same location)
  - Tree
    - Fast for searching
    - Easy to expand
    - Difficult to do in on-disk directory
- Name length
  - Fixed: easy to program
  - Variable: more flexible, better for users

Handling long file names in a directory

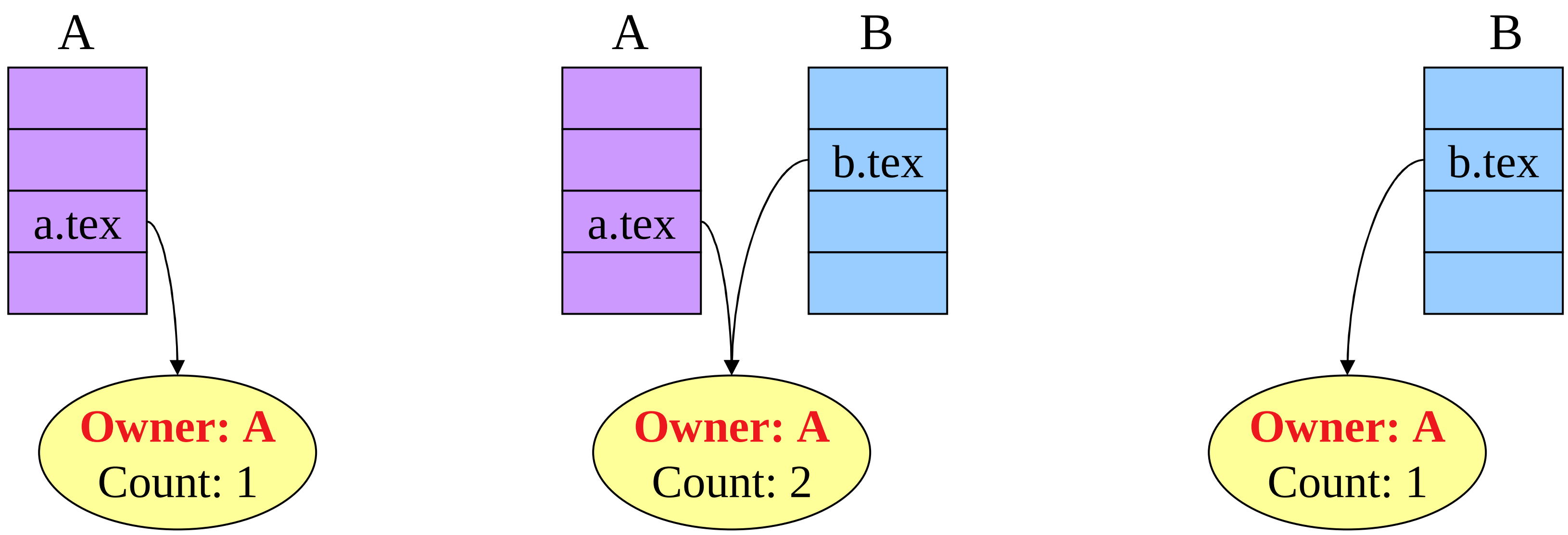


Sharing files

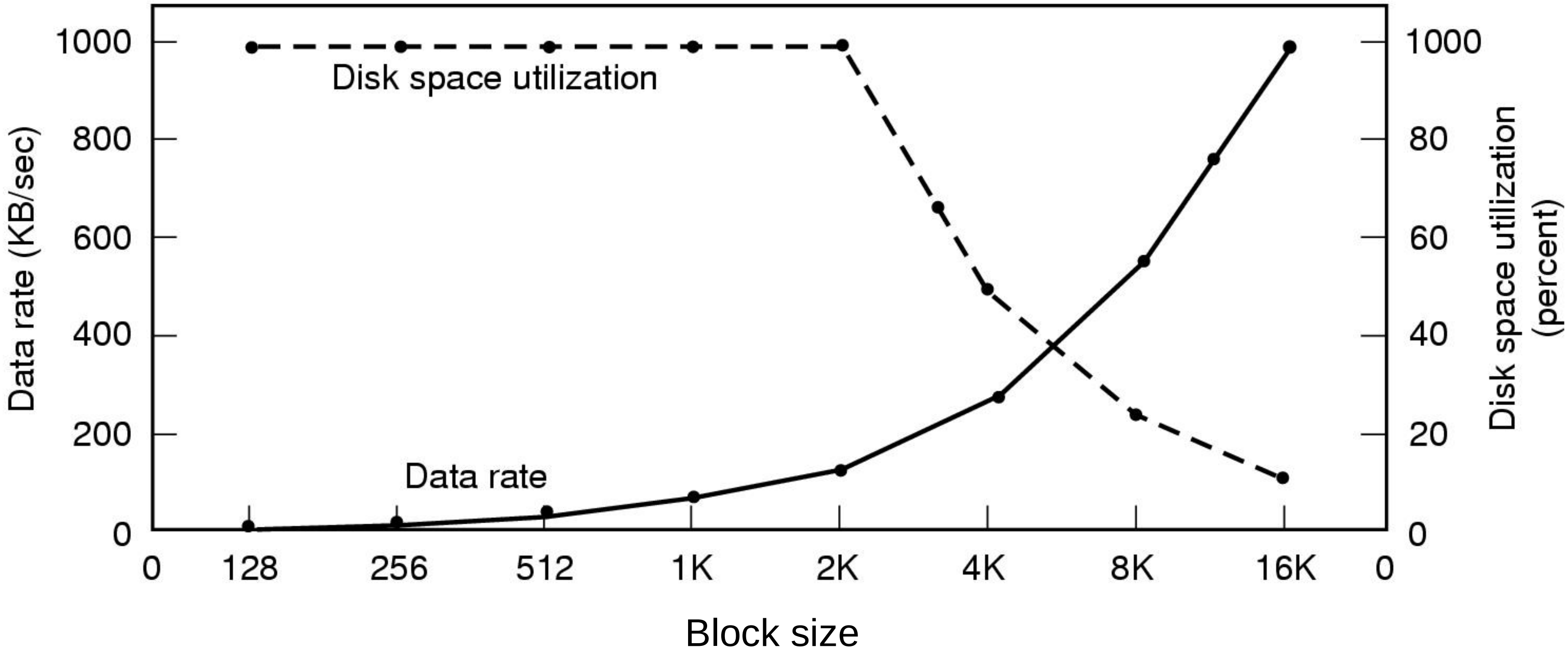


Solution: use links

- A creates a file, and inserts into her directory
- B shares the file by creating a link to it
- A unlinks the file
  - B still links to the file
  - Owner is still A (unless B explicitly changes it)

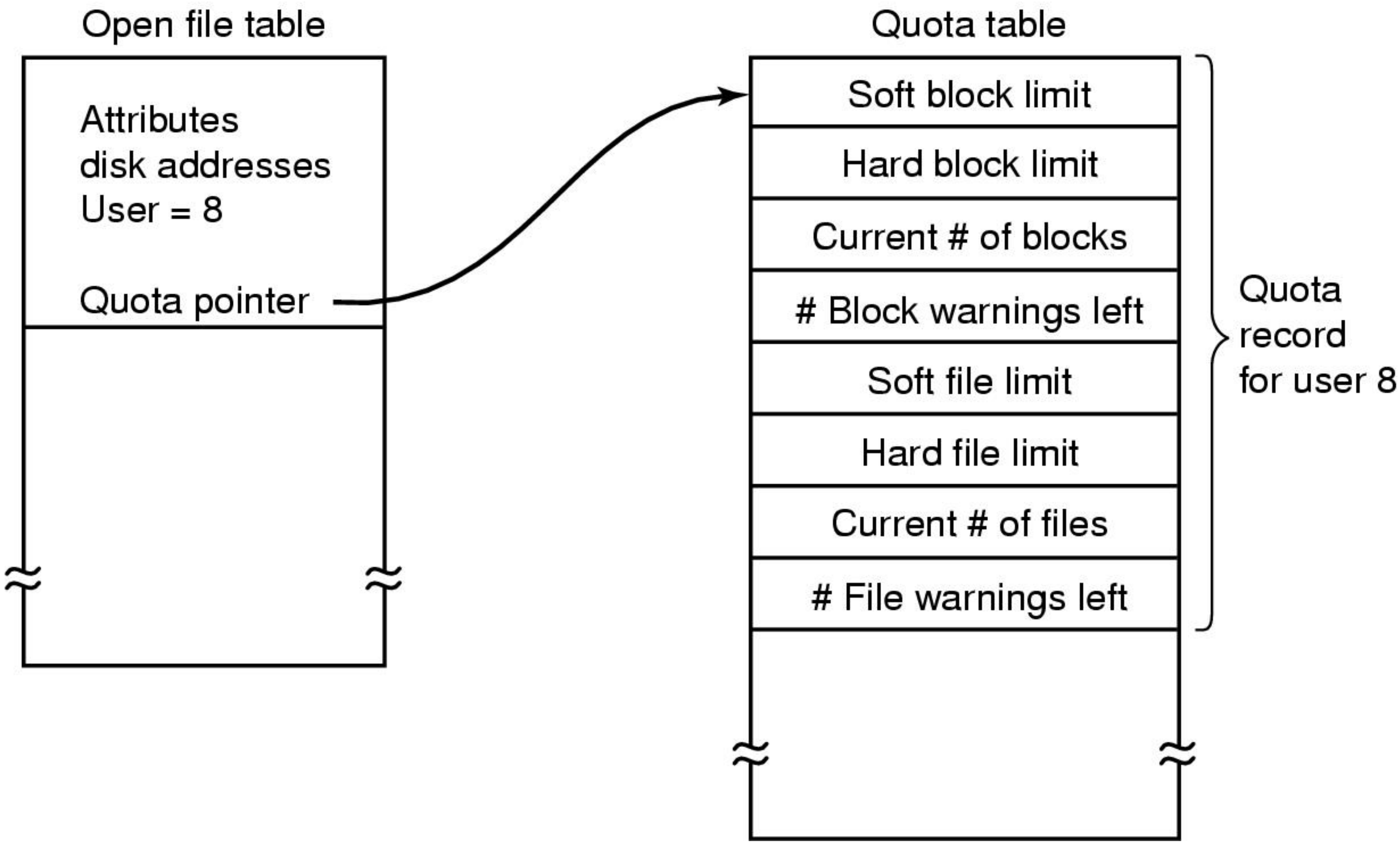


Managing disk space



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

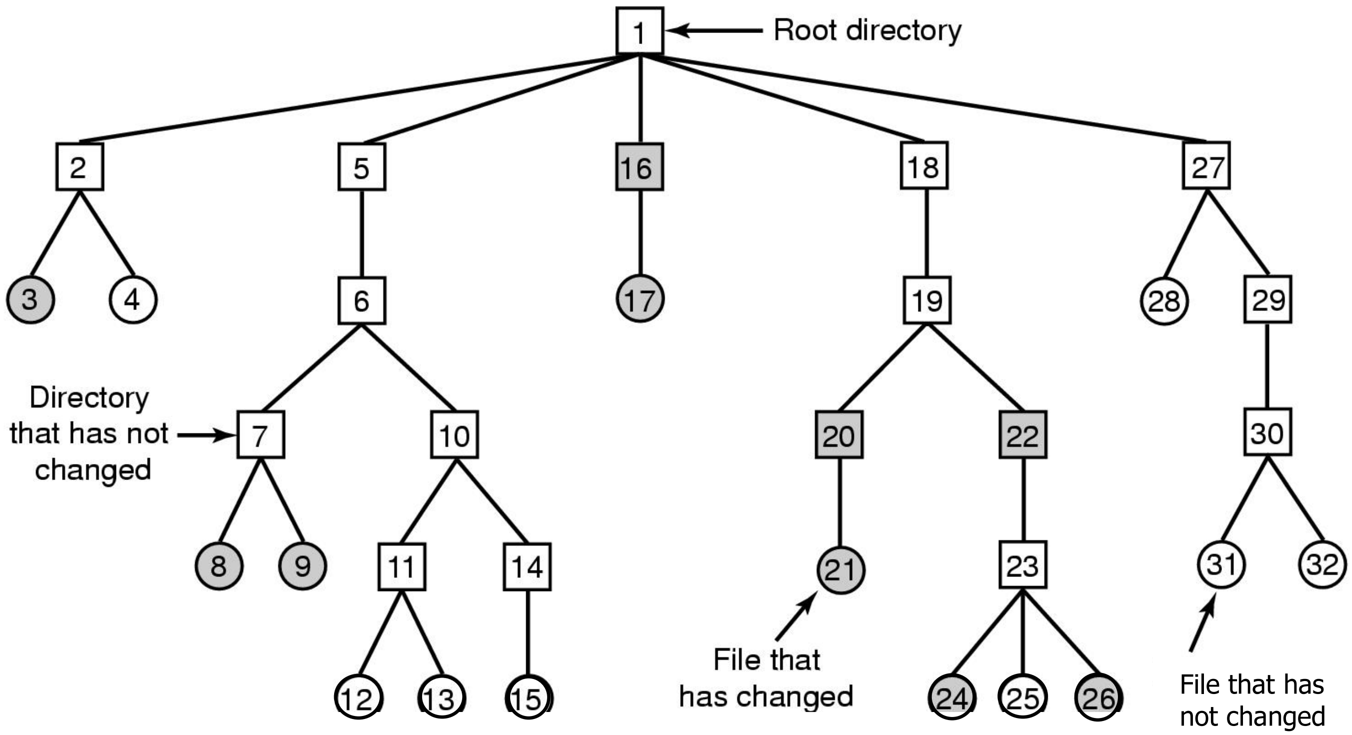
Disk quotas



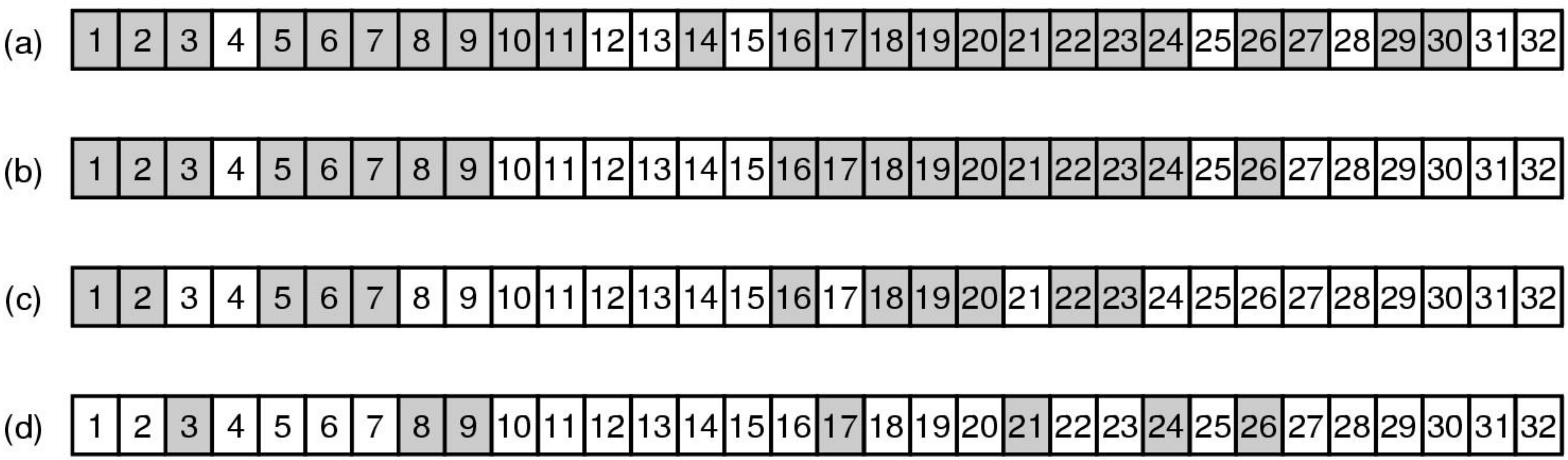
Open file table
Attributes
disk addresses
User = 8
Quota pointer
Quota table
Soft block limit
Hard block limit
Current # of blocks
Block warnings left
Soft file limit
Hard file limit
Current # of files
File warnings left
Quota
record
for user 8

Backing up a file system

- A file system to be dumped
  - Squares are directories, circles are files
  - Shaded items, modified since last dump
  - Each directory & file labeled by i-node number



Bitmaps used in a file system dump



Checking the file system for consistency

Consistent

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(a)

Missing (“lost”) block

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

(b)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	2	0	0	0	0	1	1	0	0	0	1	1	Free blocks

Duplicate block in free list

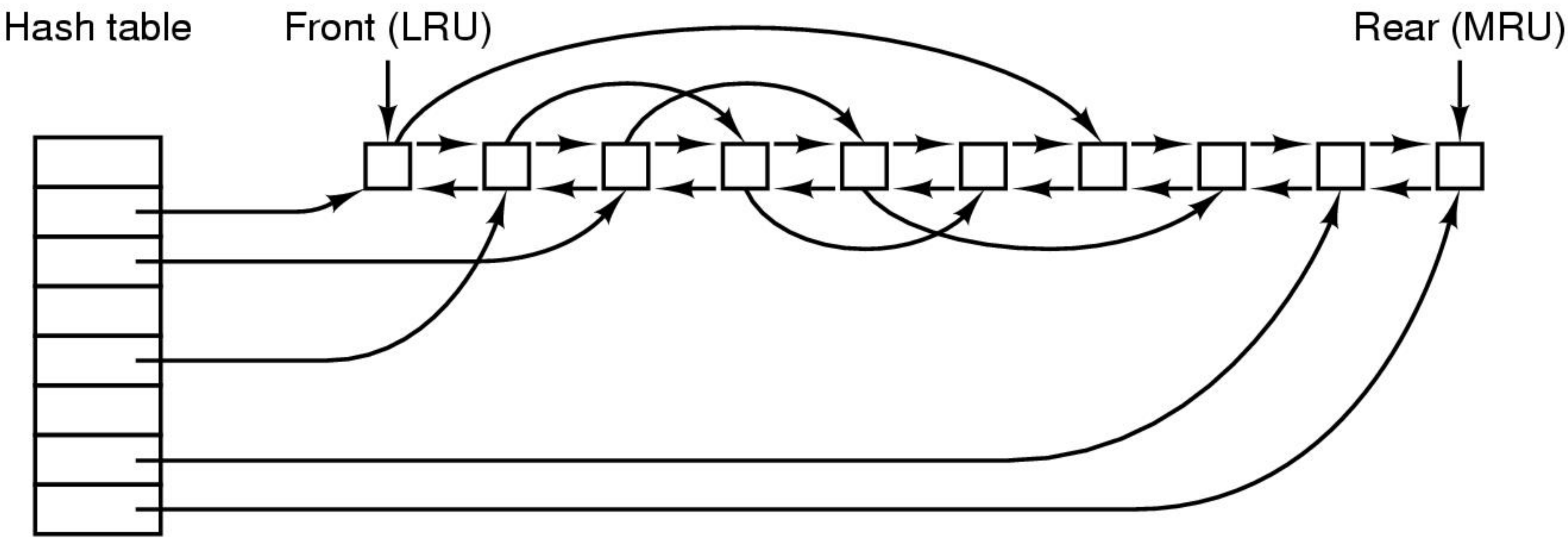
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0	Blocks in use
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1	Free blocks

Duplicate block in two files

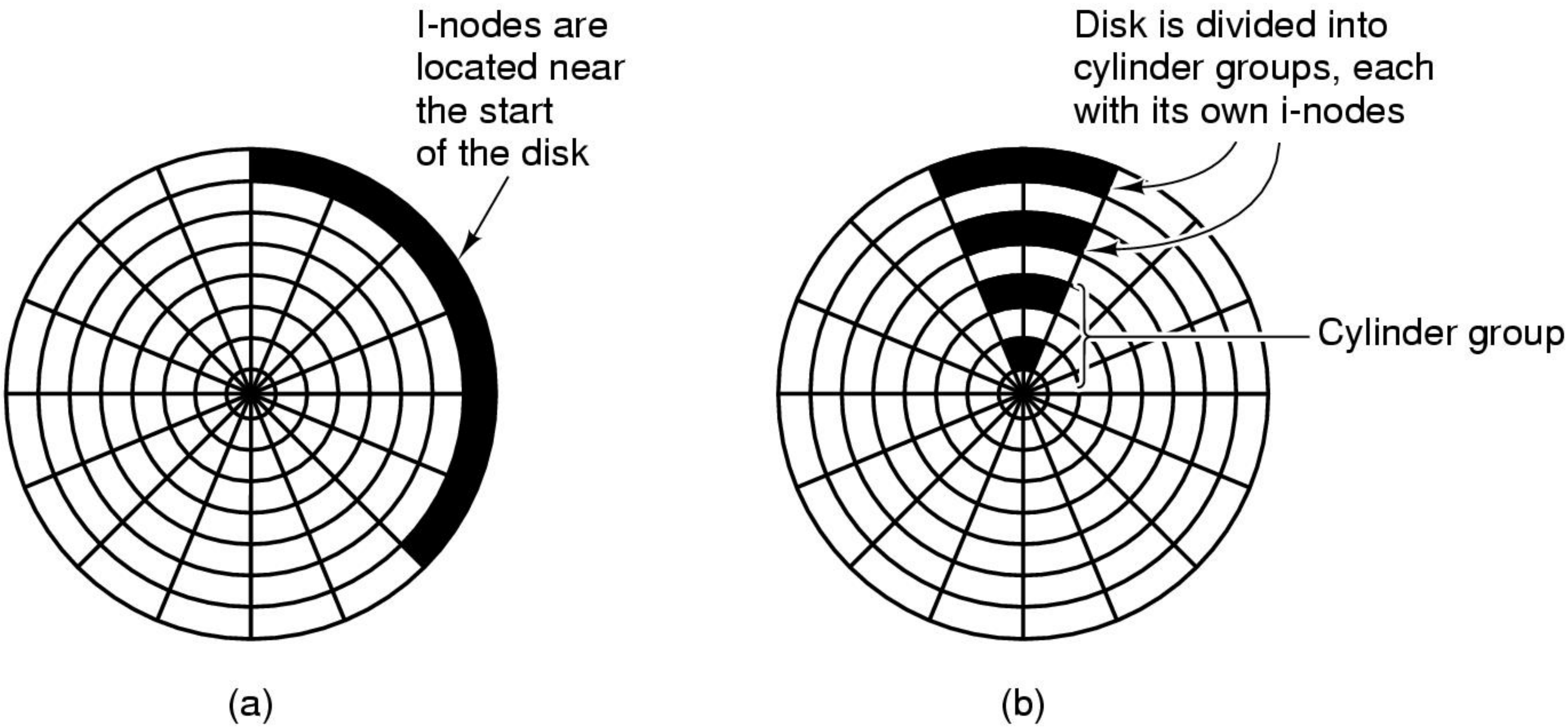
File system cache

- Many files are used repeatedly
  - Option: read it each time from disk
  - Better: keep a copy in memory
- File system cache
  - Set of recently used file blocks
  - Keep blocks just referenced
  - Throw out old, unused blocks
    - Same kinds of algorithms as for virtual memory
    - More effort per reference is OK: file references are a lot less frequent than memory references
- Goal: eliminate as many disk accesses as possible!
  - Repeated reads & writes
  - Files deleted before they're ever written to disk

File block cache data structures



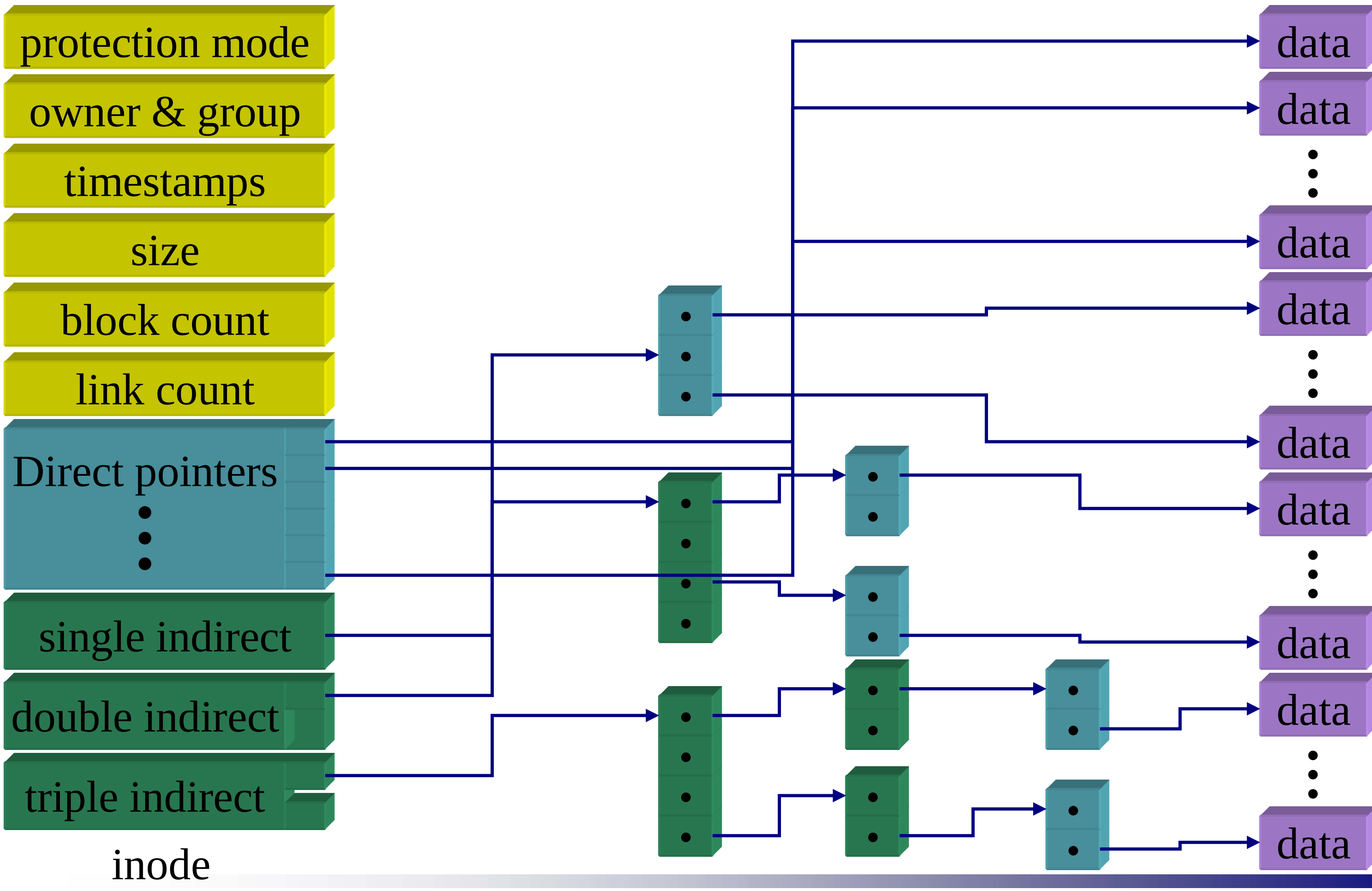
Grouping data on disk



Log-structured file systems

- Trends in disk & memory
  - Faster CPUs
  - Larger memories
- Result
  - More memory -> disk caches can also be larger
  - Increasing number of read requests can come from cache
  - Thus, most disk accesses will be writes
- LFS structures entire disk as a log
  - All writes initially buffered in memory
  - Periodically write these to the end of the disk log
  - When file opened, locate i-node, then find blocks
- Issue: what happens when blocks are deleted?

Unix **F**ast **F**ile **S**ystem indexing scheme

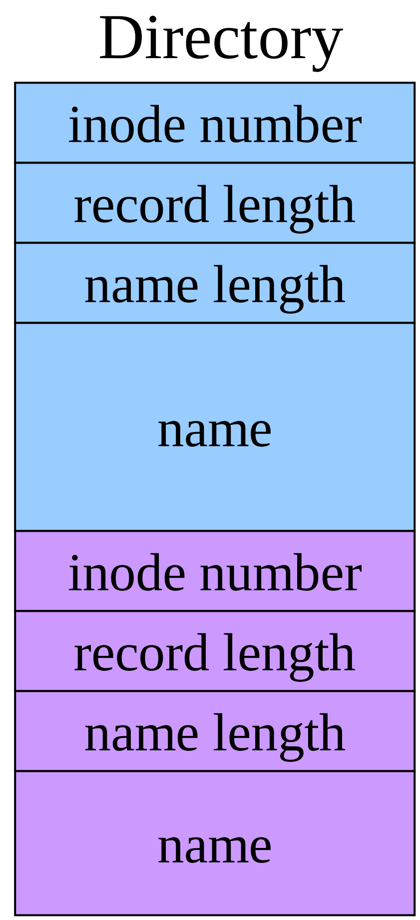


More on Unix FFS

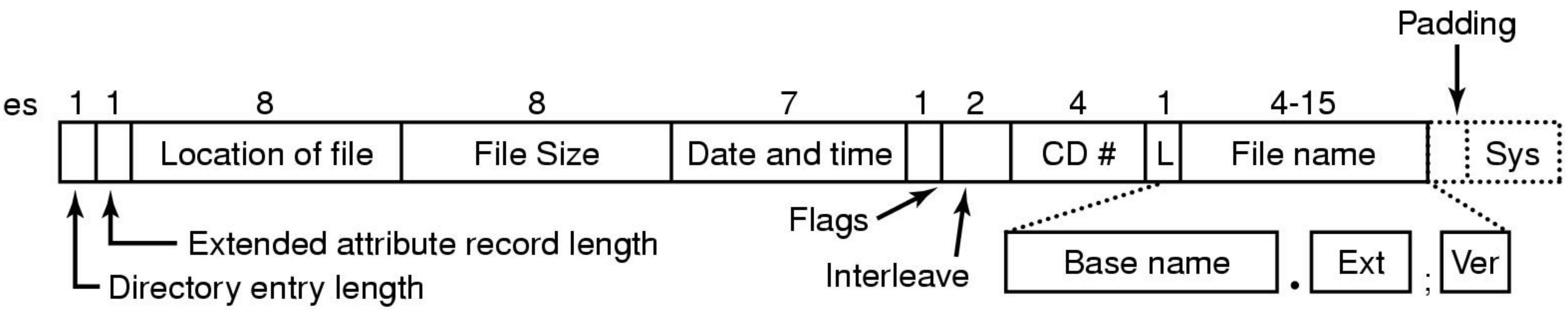
- First few block pointers kept in directory
  - Small files have no extra overhead for index blocks
  - Reading & writing small files is very fast!
- Indirect structures only allocated if needed
- For 4 KB file blocks (common in Unix), max file sizes are:
  - 48 KB in directory (usually 12 direct blocks)
  - 1024 * 4 KB = 4 MB of additional file data for single indirect
  - 1024 * 1024 * 4 KB = 4 GB of additional file data for double indirect
  - 1024 * 1024 * 1024 * 4 KB = 4 TB for triple indirect
- Maximum of 5 accesses for any file block on disk
  - 1 access to read inode & 1 to read file block
  - Maximum of 3 accesses to index blocks
  - Usually much fewer (1-2) because inode in memory

Directories in FFS

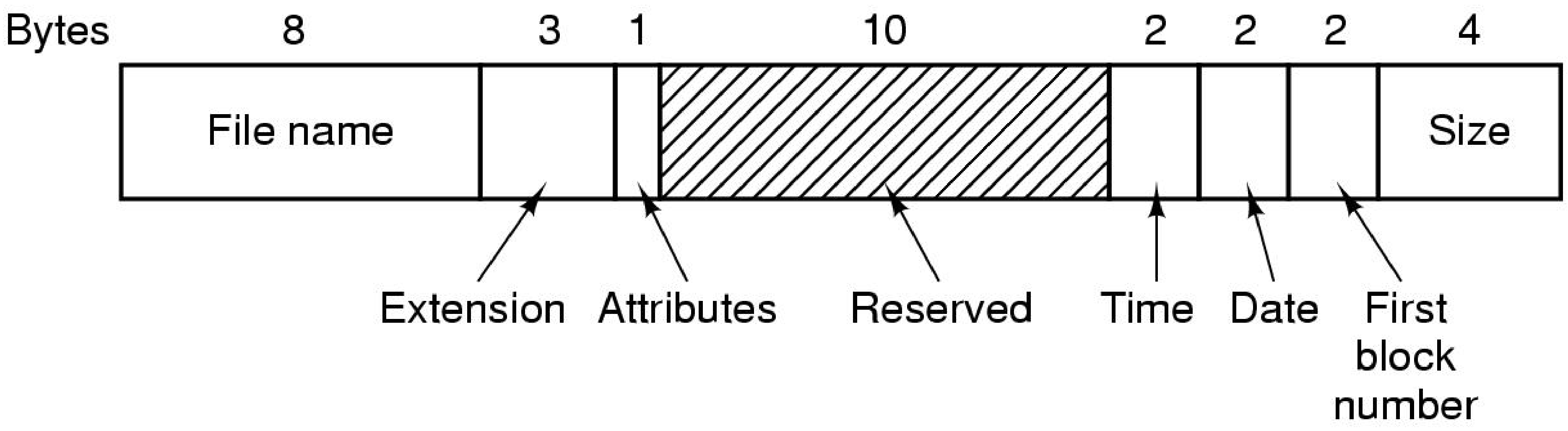
- Directories in FFS are just special files
  - Same basic mechanisms
  - Different internal structure
- Directory entries contain
  - File name
  - I-node number
- Other Unix file systems have more complex schemes
  - Not always simple files…



CD-ROM file system



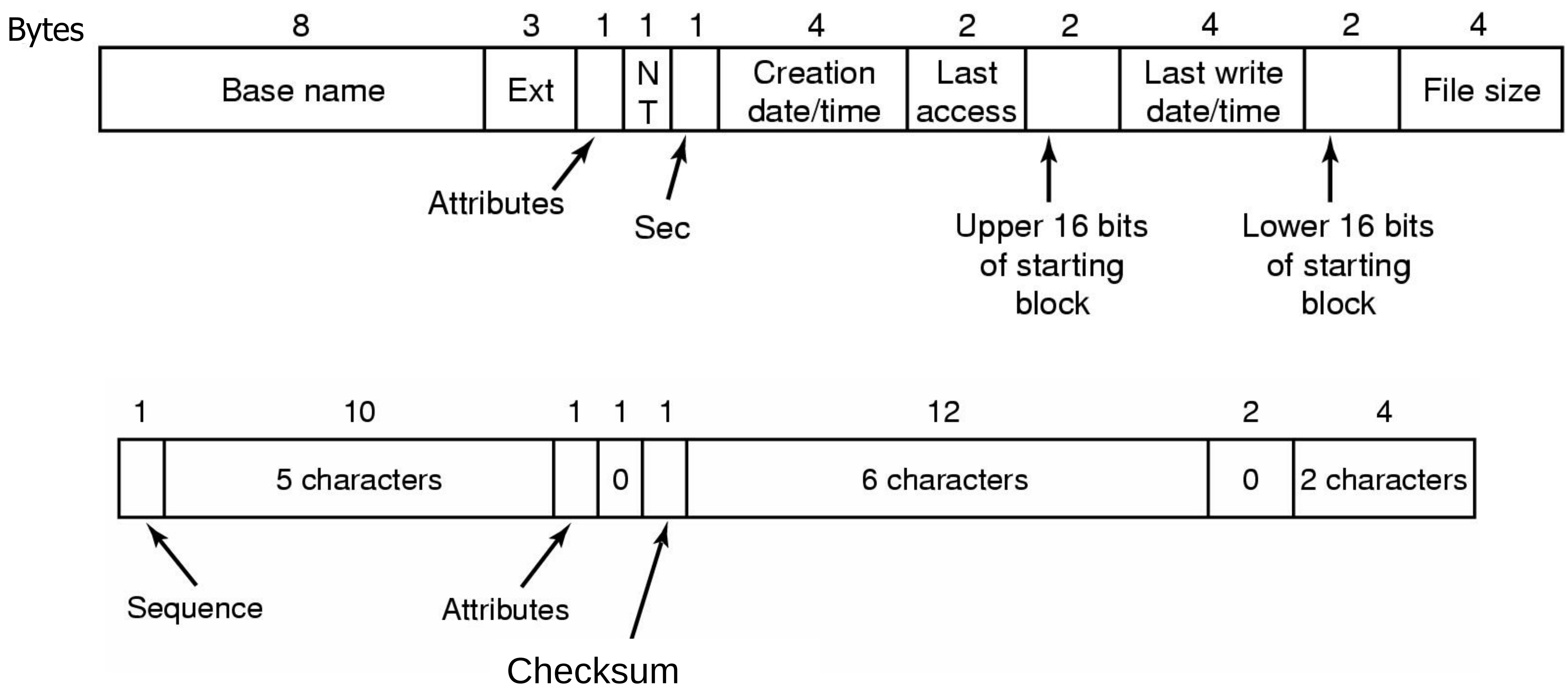
Directory entry in MS-DOS



MS-DOS File Allocation Table

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

Windows 98 directory entry & file name



Storing a long name in Windows 98

68	d o g	A	0	CK					0	
3	o v e	A	0	CK	t h	e		l a	0	z y
2	w n f o	A	0	CK	x	j	u	m p	0	s
1	T h e q	A	0	CK	u i	c	k	b	0	r o
T	H E Q U I ~ 1	A	NT	S	Creation time	Last acc	Upp	Last write	Low	Size

Bytes

- Long name stored in Windows 98 so that it's backwards compatible with short names
  - Short name in "real" directory entry
  - Long name in "fake" directory entries: ignored by older systems
- OS designers will go to great lengths to make new systems work with older systems…