25 Example File Systems

Special Features of Files
Example File Systems

February 9 2009
Winter Term 2008/09
Gerd Liefländer

Recommended Reading

Bacon, J.:	Concurrent Systems (5)
Nehmer, J.:	Systemsoftware: Grlg. mod. BS, (9)
Silberschatz, A.:	Operating System Concepts (10,11)
Stallings, W.:	Operating Systems (12)
Tanenbaum, A.:	Modern Operating Systems (5, 6)

Roadmap for Today

- Special Features of Files & File Systems
    - File Control Structures
    - Memory Mapped Files
    - Log Structured FS
- Example File Systems
    - Unix
    - BSD FFS
    - EXT2
    - Linux VFS
    - Reiser FS

File Control Structures

File Control Block

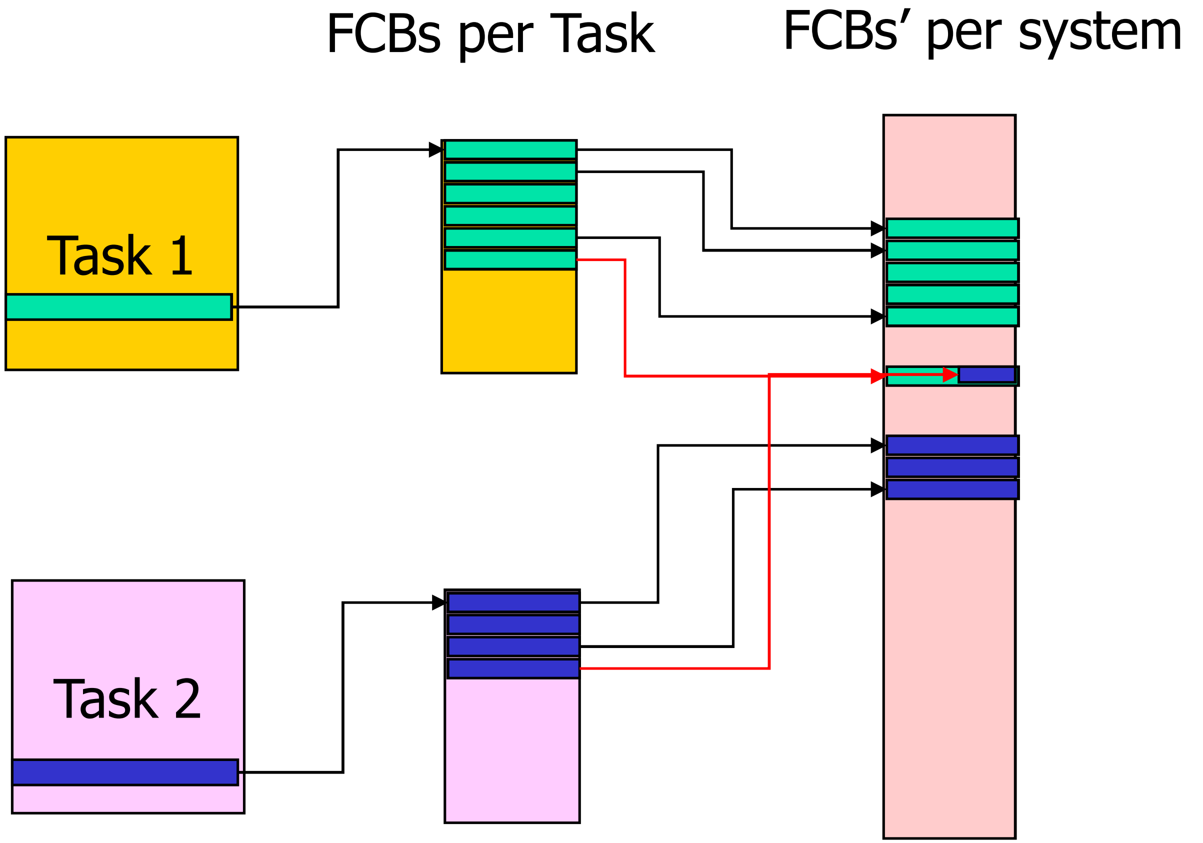
- Per application there is a list of opened files
- Per opened file there is a file control block (FCB)
  - Position pointer
  - Current block address
  - Links to buffers in main memory
  - Filling grade of buffer
  - Lock information
  - Access dates (e.g. time when file was opened)
  - Access rights

Unix FCBs per Task/Process

- Per default each task has a couple of standard files
  - `stdin`   `FID = 0`
  - `stdout`   `FID = 1`
  - `stderr`   `FID = 2`
  - `FIDs` with higher value are used for other files
  - Once a file is closed, its FID is never used again
- With FIDs it is easy to redirect output to different files and to establish piped applications

Unix FCBs

- Besides collecting info on opened files per task, in most system there is also a table/list of all opened files



Memory Mapped Files

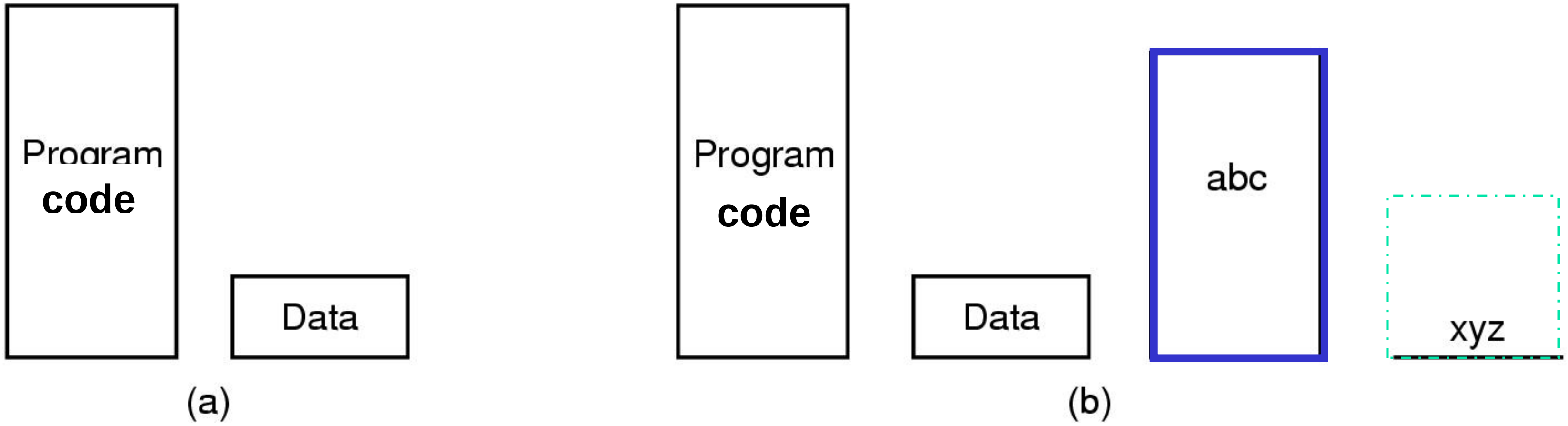
Memory Mapped Files

- Map a file into a region of an AS of a task
- Idea: Access a mapped file as any other AS region

Implementation:

- Reserve appropriate region within AS *and then?*
  - PTEs point to file disk blocks instead of …?
- Via page fault you load the corresponding “file page”
- Upon unmap, write back all modified pages

Memory Mapped Files



(a) Segmented process before mapping files into its address space

(b) Process after mapping
  - existing file **abc** into one segment
  - reserving a new segment for a new file **xyz**

Memory Mapped Files

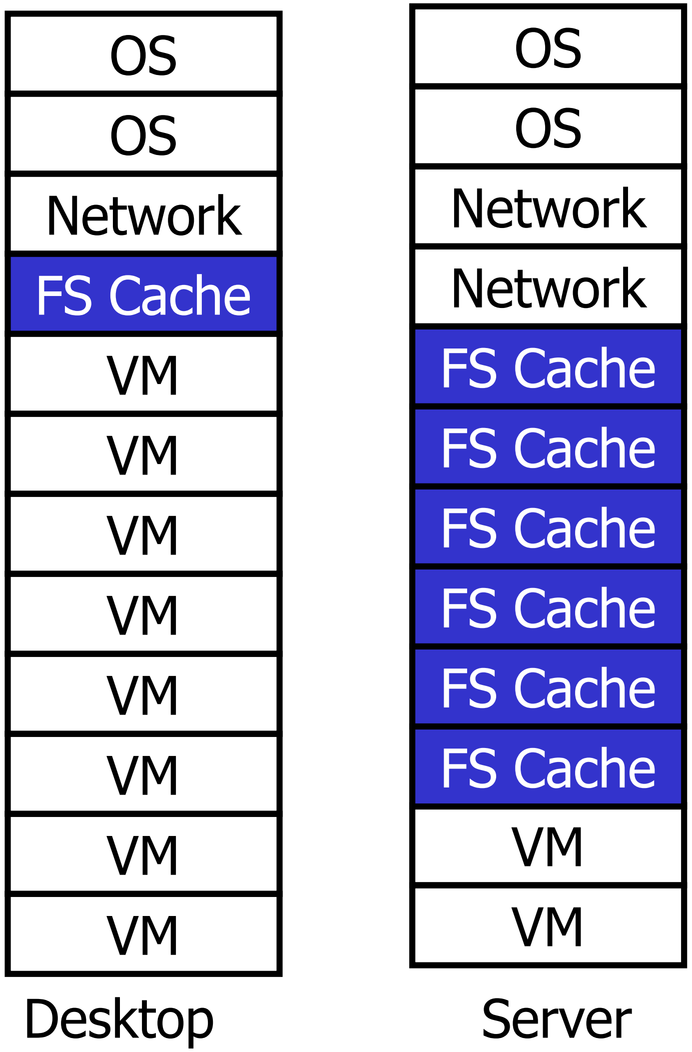
- Avoids translating from disk format to RAM format (and vice versa)
  - Supports complex structures
  - No read/write system calls!!!
  - Unmap the file implicitly when task/process exits
- Problems:
  - Determining actual size after several modifications
  - Care must be taken if file `f` is shared, e.g.
    - process P_1 uses `f` as a memory-mapped file
    - process P_2 uses `f` via conventional file operations (read/write)

Memory Mapped Files

- Appended slides are by Vivek Pai et al.
- Another set of good slides concerning memory-mapped files in Linux
- see: Linux Proseminar 2004/04 + 2004/05
    - Fabian Keller
    - Sebastian Möller
- Bad news: Study of your own!!!!
- Good news: Not in the focus of this year's exams

Allocating Memory

- Old days:
    - manual tuning of sizes
- Benefits?
- Drawbacks?



Manual Memory Tuning

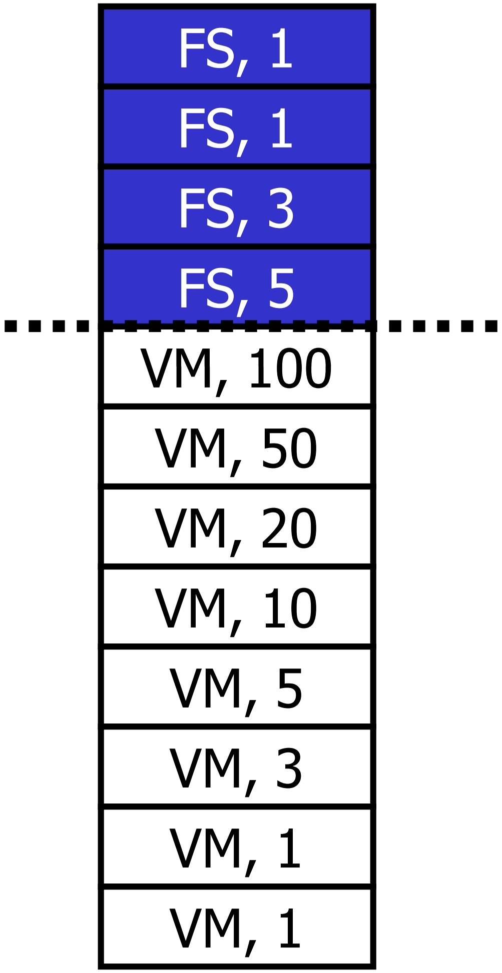
- Fixed-size allocations for VM, FS cache
- Done right, protects programs from each other
    - Backing up file system trashes FS cache
    - Large-memory programs don’t compete with disk-bound programs
- However, done poorly ⇒ memory underutilized

What Is Main Memory?

- At some level, a cache for the disk
  - Permanent data written back to fs
  - Temporary data in main memory or swap
- Main memory is much faster than disk
- Consider one program that accesses lots of files and uses lots of memory
  - How do you optimize this program?
  - Could you view all accesses as page faults?

Consider Ages With Pages

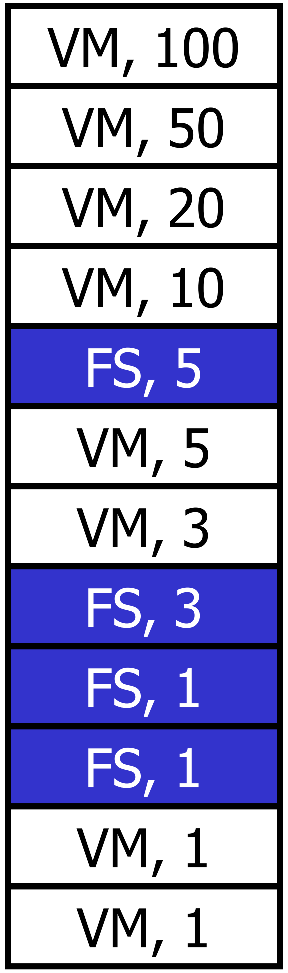
- *What happens if 5 FS pages are really active?*
- *What happens if relative demands change over time?*



FS, 1
FS, 1
FS, 3
FS, 5
VM, 100
VM, 50
VM, 20
VM, 10
VM, 5
VM, 3
VM, 1
VM, 1

Unified VM Systems

- *Now what happens when a page is needed?*
- *What happens on disk backup?*
- *Did we have the same problem before?*



VM, 100
VM, 50
VM, 20
VM, 10
FS, 5
VM, 5
VM, 3
FS, 3
FS, 1
FS, 1
VM, 1
VM, 1

Why Mmap?

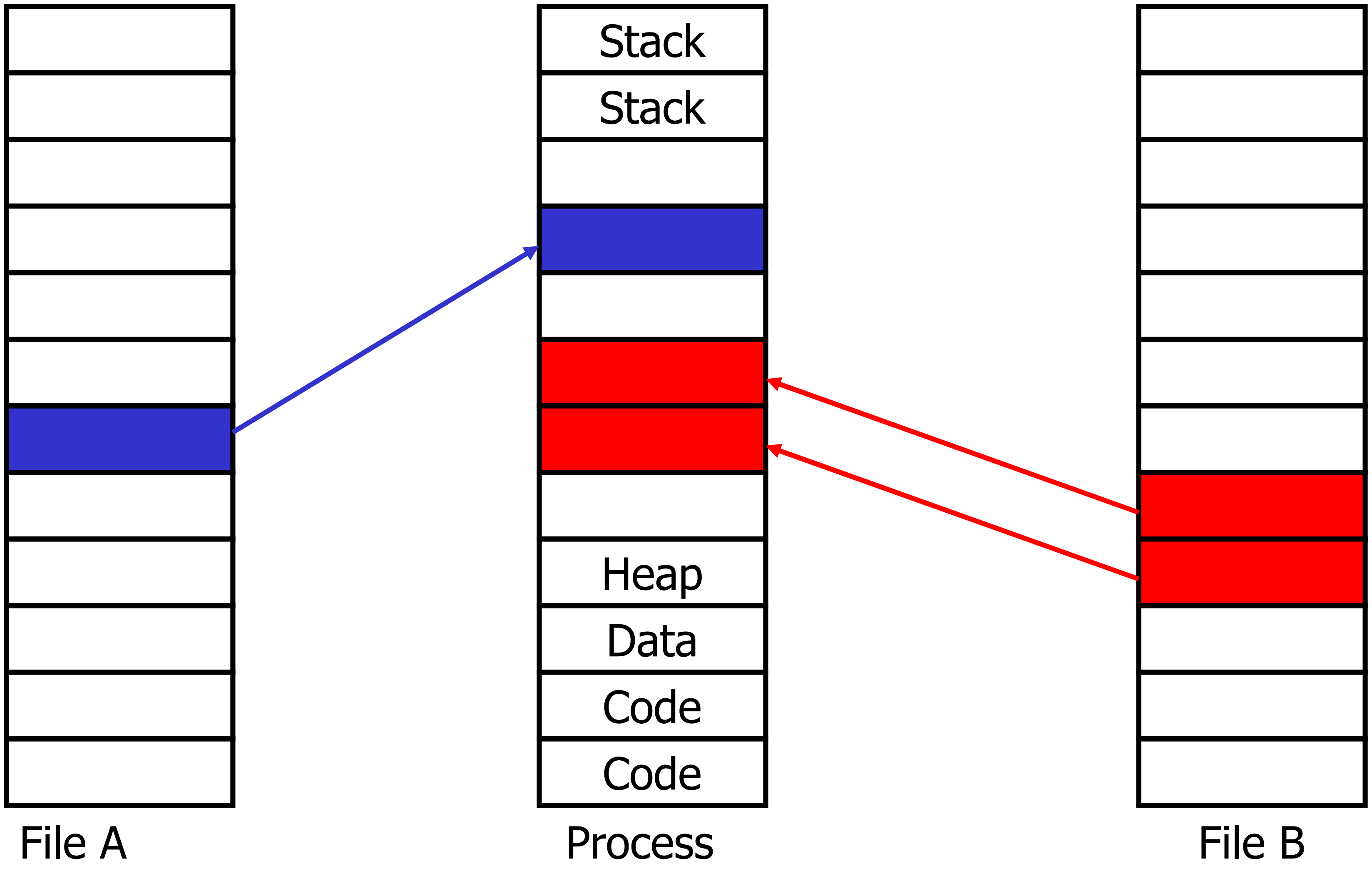
- File pages are a lot like VM pages
- We don’t load all of a process at once
- *Why load all of a file at once?*
- *Why copy a file to access it?*
  - There’s one good reason

Mmap Definition

```
void *mmap(void *addr, size_t len,
           int prot, int flags,
           int fildes, off_t off);
```

- `addr`: where we want to map it (into our AS)
- `len`: how much we want mapped
- `prot`: allow reading, writing, exec
- `flags`: is mapped shared/private/anonymous, fixed/variable location, swap space reserved?
- `fildes`: what file is being mapped
- `off`: start offset in file

Mmap Diagram



Mmap Implications

- # of VM regions increases
  - Was never really just code/text/heap/stack
  - Access/protection info on all regions
- File system no longer sole way to access file
  - Previously, access info via read( ) and write( )
  - Same file via file system and mmap?

Mmap Versus Read

- When read( ) completes
  - All pages in range were loaded at some point
  - A copy of the data in user’s buffers
  - If underlying file changes, no change to data in user’s buffer
- When mmap( ) completes
  - Mapping of the file is complete
  - Virtual address space modified
  - No guarantee file is in memory

Cost Comparison

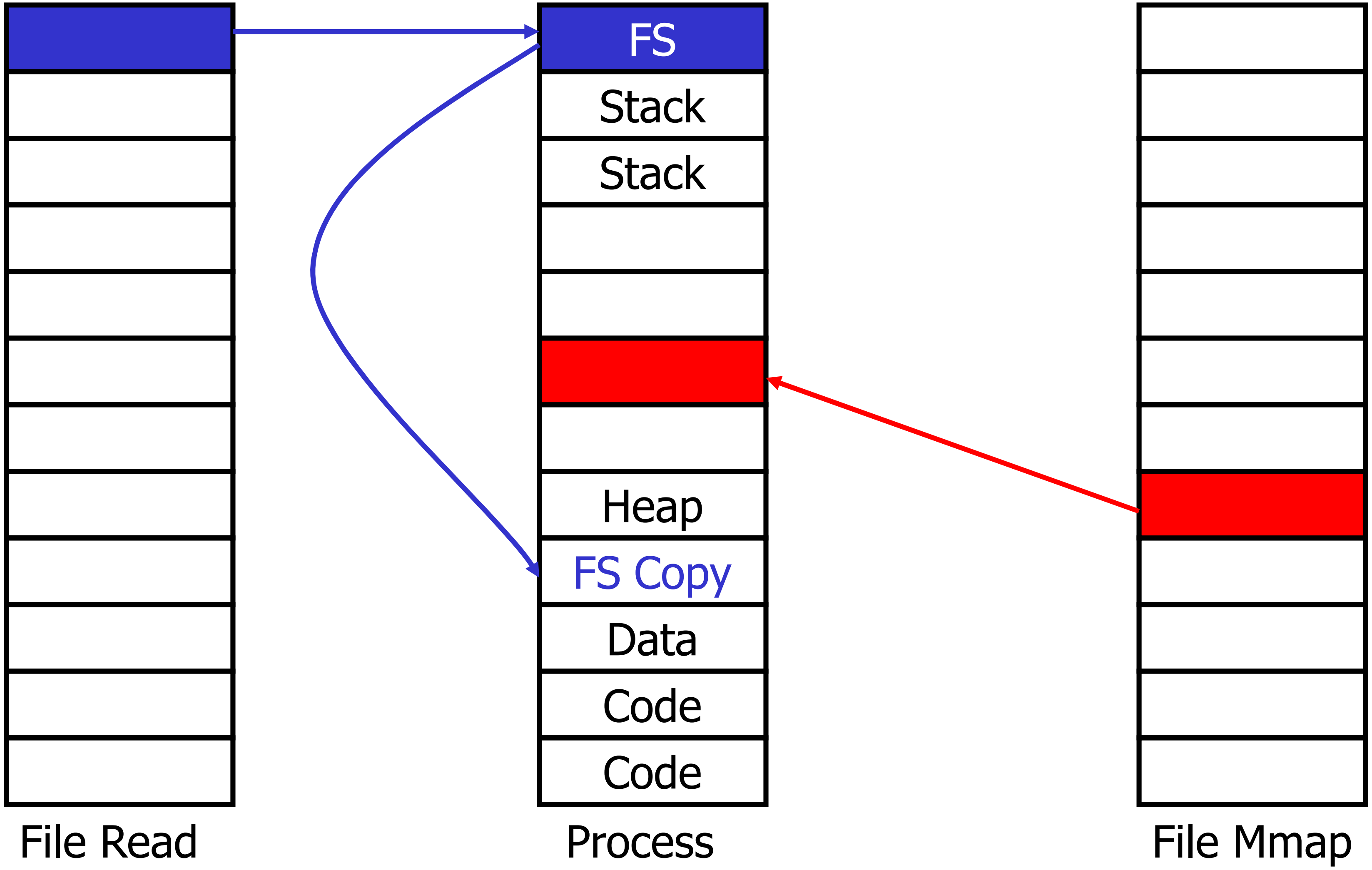
- Read:
    - All work done (incl disk) before call returns
    - No extra VM trickery needed
    - Contrast with write( )
- Mmap:
    - Inode in memory from open( )
    - Mapping is relatively cheap
    - Pages needed only on access

Lazy Versus Eager

- Eager:
  - Do it right now
  - Benefit: low latency if you need it
  - Drawback: wasted work if you don’t

- Lazy:
  - Do it at the last minute
  - Benefit: “pay as you go”
  - Drawback: extra work if you need it all

Double Buffering



Sharing Memory

- Two processes map same file shared
- Both map it with “shared” flag
  - Same physical page accessed by two processes at two virtual addresses
- What happens when that page victimized (PTE mechanics)?
  - Have we seen this somewhere else?

Reloading State

- Map a file at a fixed location
- Build data structures inside it
- Re-map at program startup
- Benefits versus other approaches?

What Is a “Private” Mapping?

- Process specifies changes not to be visible to other processes
- Modified pages look like VM pages
    - Written to swap if pressure
    - Disposed when process dies

Log Structured FS

Log-Structured File Systems*

- With CPUs faster & memory larger ⇒
- disk caches can also be larger ⇒
- many read requests can come from cache
- most disk accesses will be writes
    - If writes, will cover only a few bytes
    - If writes, to Unix-like new files
        - Inode of directory, directory
        - Inode of file, meta blocks and data blocks of file

*Rosenblum and Ousterhout

Log-Structured File Systems

- Log-structured FS: use disk as a circular buffer:
- Write all updates, including inodes, meta data to end of log
  - have all writes initially buffered in memory
  - periodically write these within 1 segment (1 MB)
  - when file opened, locate i-node, then find blocks
- From the other end, clear all data, no longer used

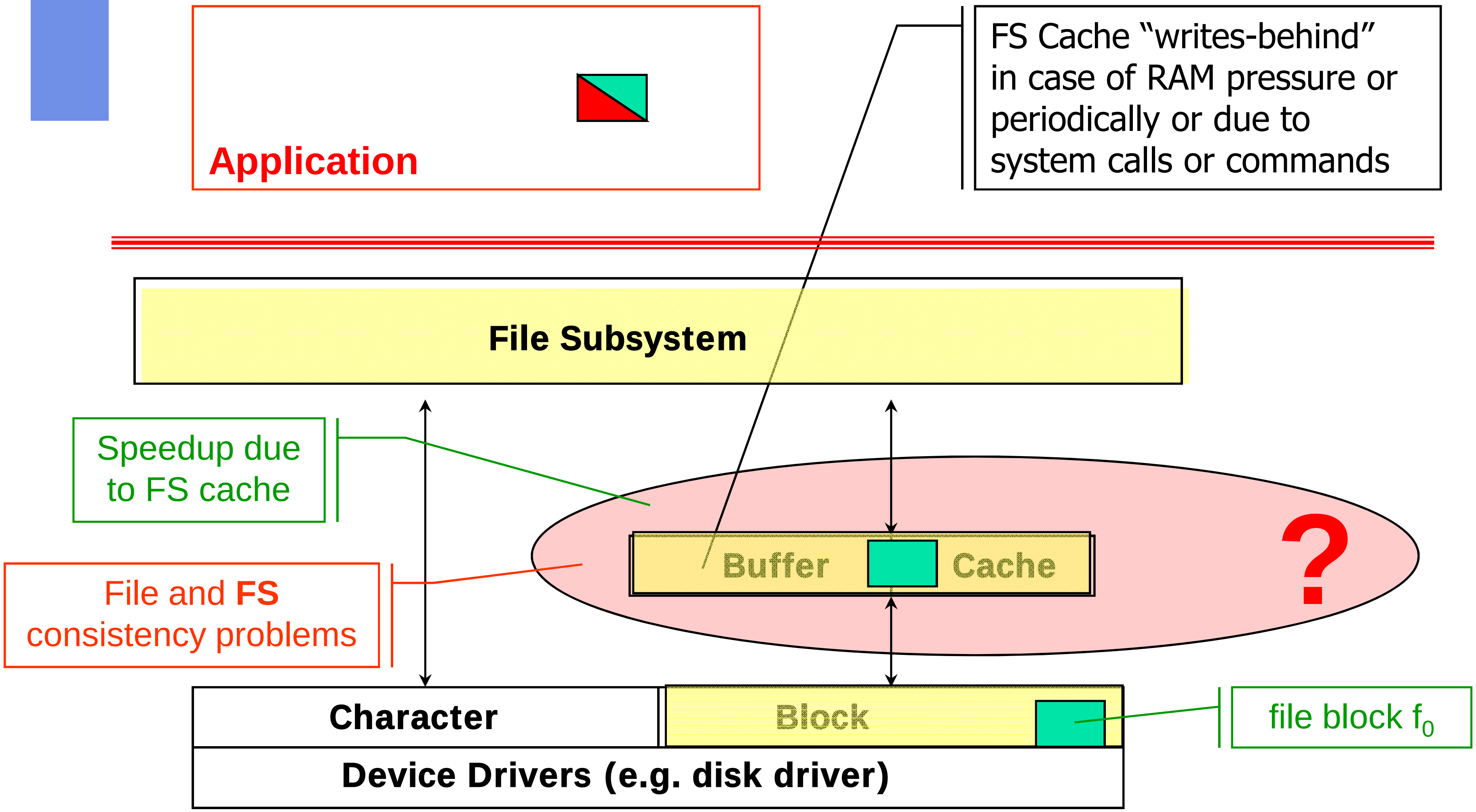
Example File Systems

- CD-ROM
- Classic FS
  - CP/M
  - MS-DOS
  - Unix
  - BSD FFS
  - EXT2
  - Linux VFS
- Journaling (log strutured) FS
  - *EXT3*
  - *XFS*
  - *JFS*
  - Reiser
  - NTFS
  - *Veritas*

} Special slides File_Appendix1

Comparison of ~ 60 File Systems
http://en.wikipedia.org/wiki/Comparison_of_file_systems

UNIX File System Structure



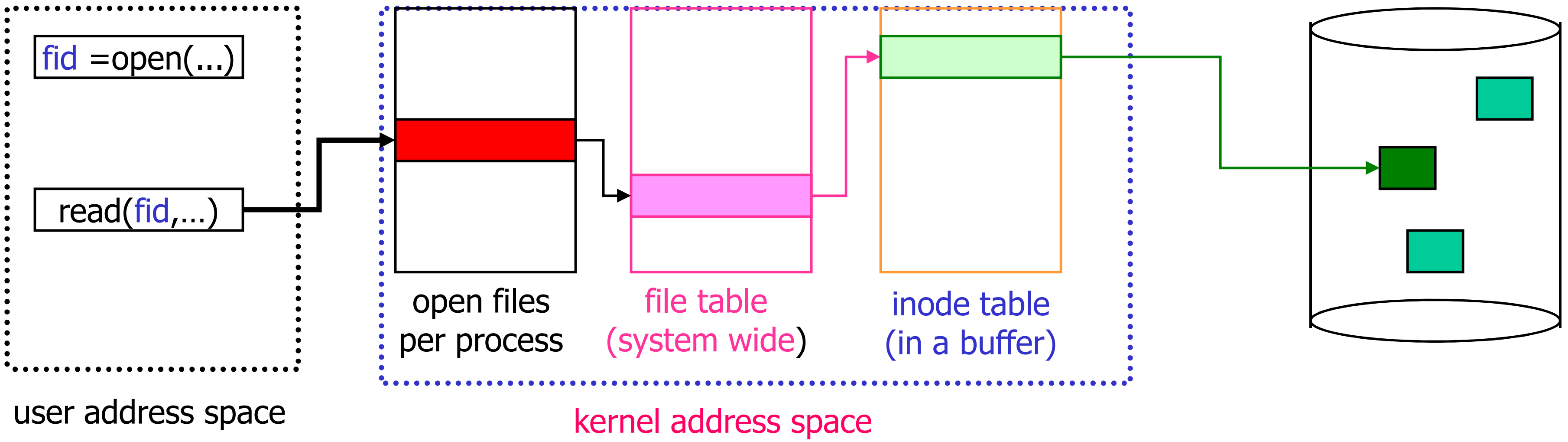
FS Cache "writes-behind" in case of RAM pressure or periodically or due to system calls or commands

Speedup due to FS cache

File and **FS** consistency problems

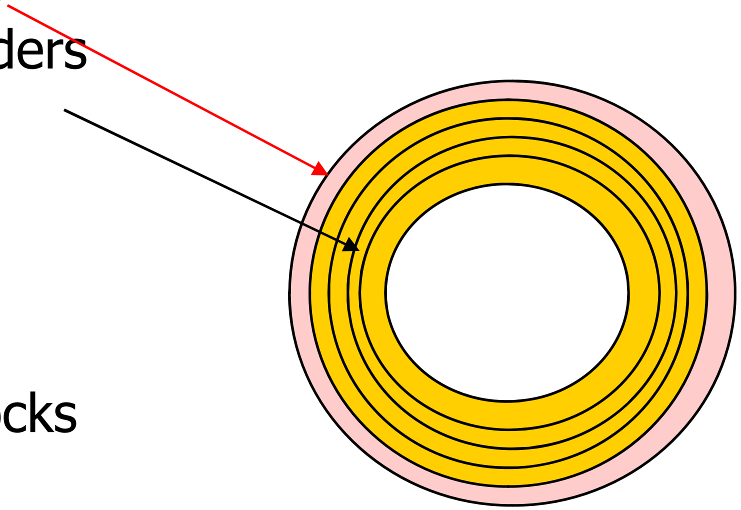
Using a Unix File

- Opening a file creates a file descriptor fid
- Used as an index into a process-specific table of open files
- The corresponding table entry points to a system-wide file table
- Via buffered inode table, you finally get the data blocks



Original Unix File System

- Simple disk layout
  - Block size = sector size (512 bytes)
  - Inodes on outermost cylinders[1]
  - Data blocks on the inner cylinders
  - Freelist as a linked list
- Issues
  - Index is large
  - Fixed number of files
  - Inodes far away from data blocks
  - Inodes for directory not close together
  - Consecutive file blocks can be anywhere
  - Poor bandwidth (20 KB/sec) for sequential access



[1] in very early Unix FSs inode table in the midst of the cylinders

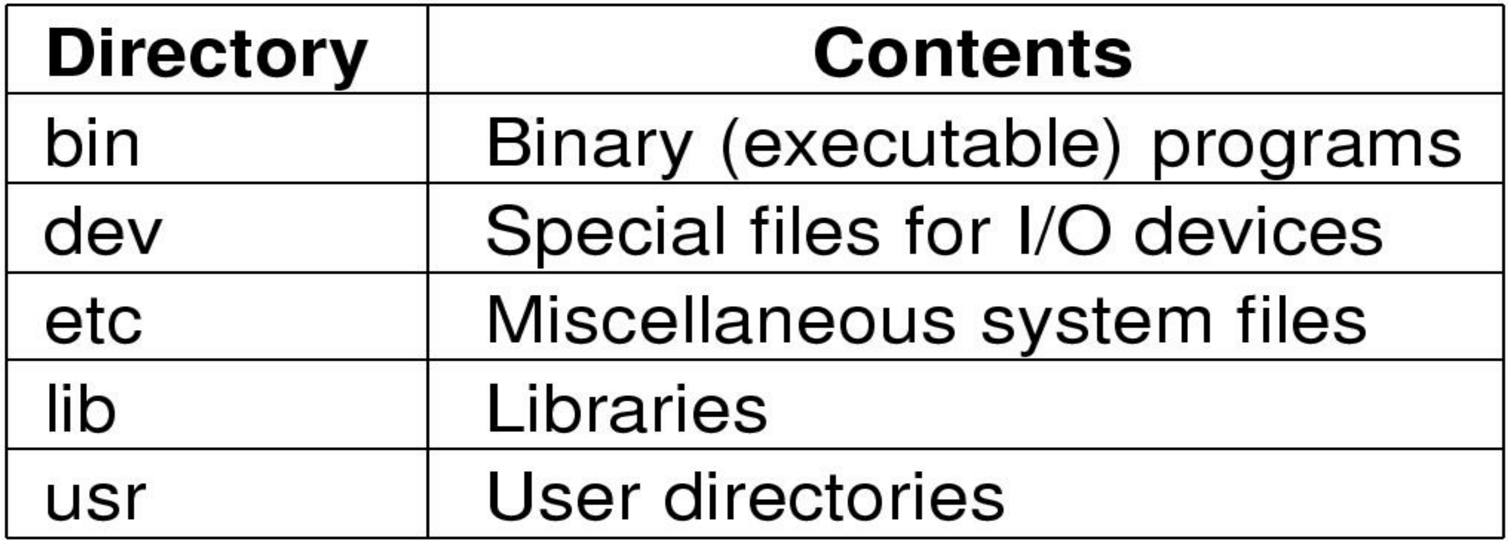
Unix File Names

- Historically only 14 characters
- Version V up to 255 ASCII characters

  <filename> . <extension>

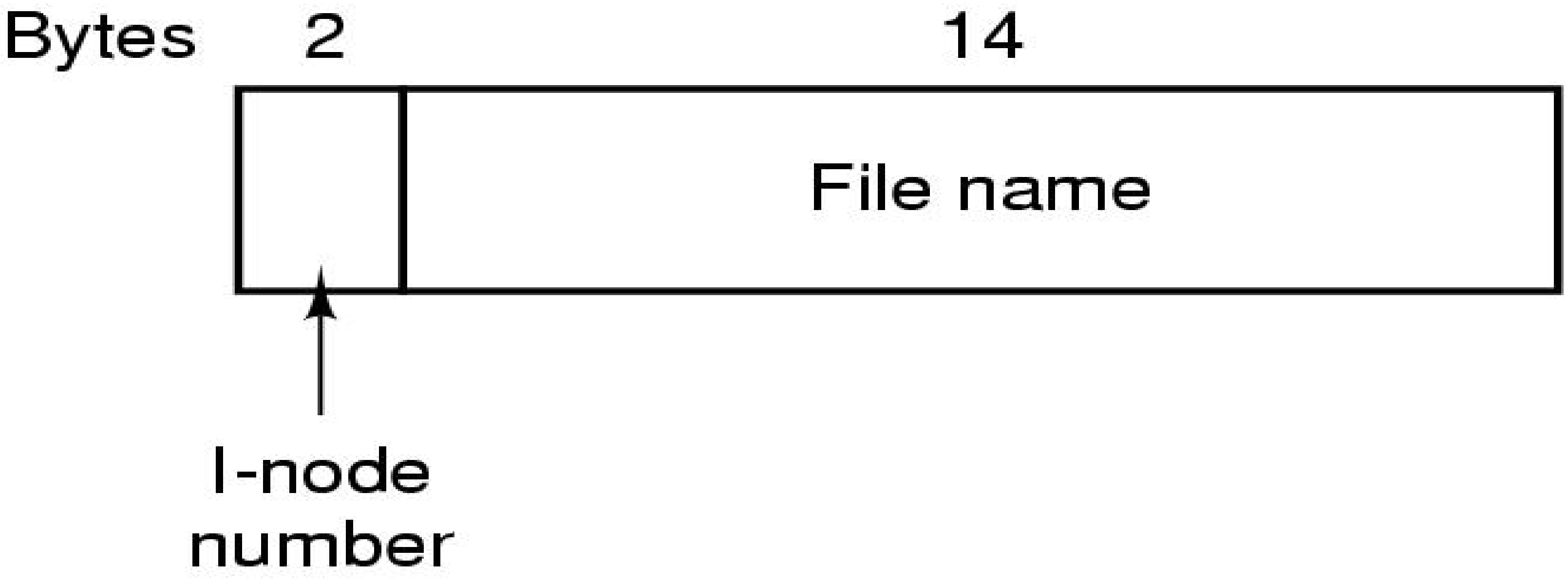
- program.c ~ a C-source code
- program.h ~ header file for type definition etc.
- program.o ~ an object file

Important Unix Directories



Directory	Contents
bin	Binary (executable) programs
dev	Special files for I/O devices
etc	Miscellaneous system files
lib	Libraries
usr	User directories

Unix V Directory Entry V7[1]



[1] Historical version

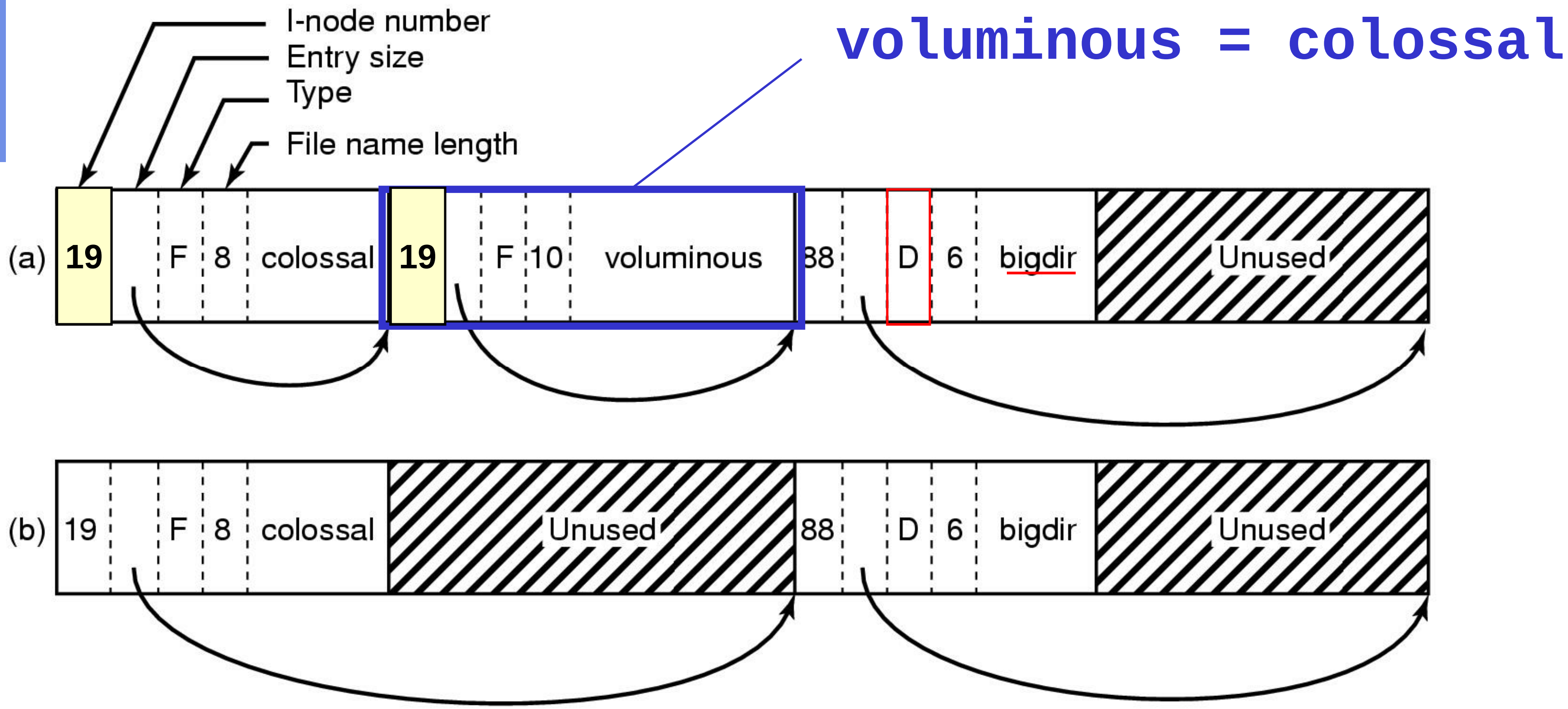
BSD FFS

- Use a larger block size: 4 KB or 8 KB
  - Allow large blocks to be chopped into fragments
  - Used for little files and pieces at the ends of files
- Use *bitmap* instead of a free list
  - Try to allocate more contiguously
  - 10% reserved disk space

BSD FFS Directory

- Directory entry needs three elements:
    - length of dir-entry (variable length of file names)
    - file name (up to 255 characters)
    - inode number (index to a table of inodes)
- Each directory contains at least two entries:
    - .. = link to the parent directory (forming the directory tree)
    - . = link to itself
- FFS offers a “tree-like structure” (like Multics), supporting human preference, ordering hierarchically

Unix BSD FFS Directory (2)



- BSD directory three entries (voluminous = hardlink to colossal)
- Same directory after file **voluminous** has been removed

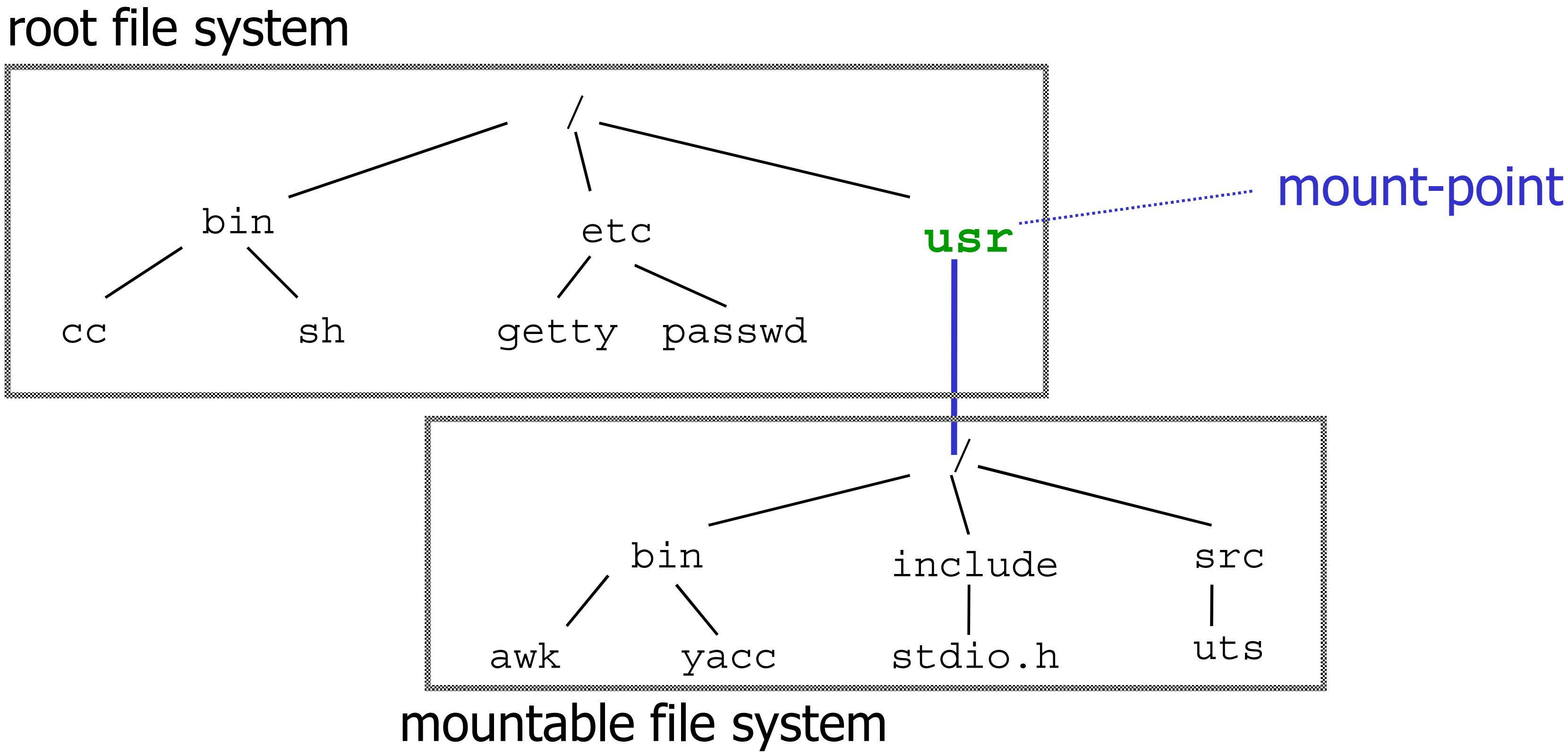
Unix Directories

- Multiple directory entries may point to same inode (hard link)
- Pathnames are used to identify files

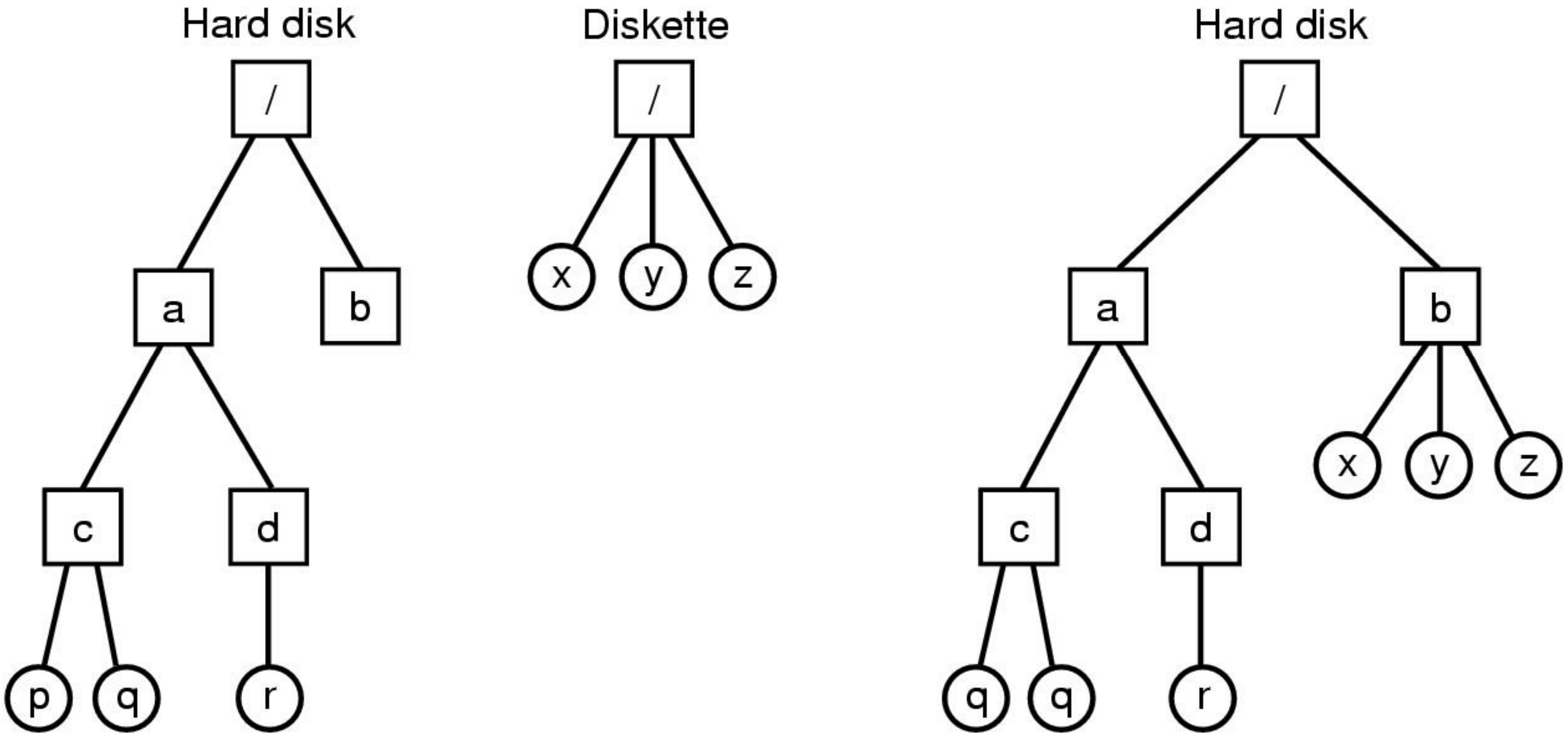
`/etc/passwd`	an absolute pathname
`../home/lief/examination`	a relative pathname

- Pathnames are resolved from left to right
- As long as it's not the last component of the pathname, the component name must be a directory
- With symbolic links you can address files and directories with different names. You can even define a symbolic link to a file currently not mounted (or even that never existed); i.e. a symbolic link is a file containing a pathname

Logical and Physical File System



Mounting a File System



(a) Before mounting

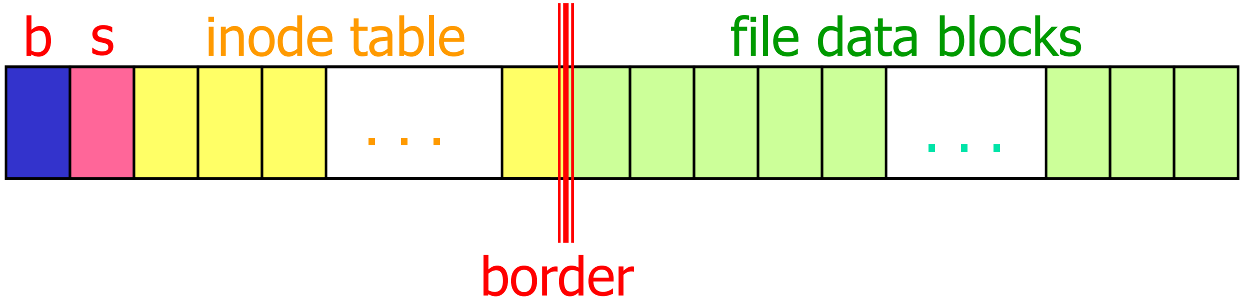
(b) After mounting

Logical and Physical File System

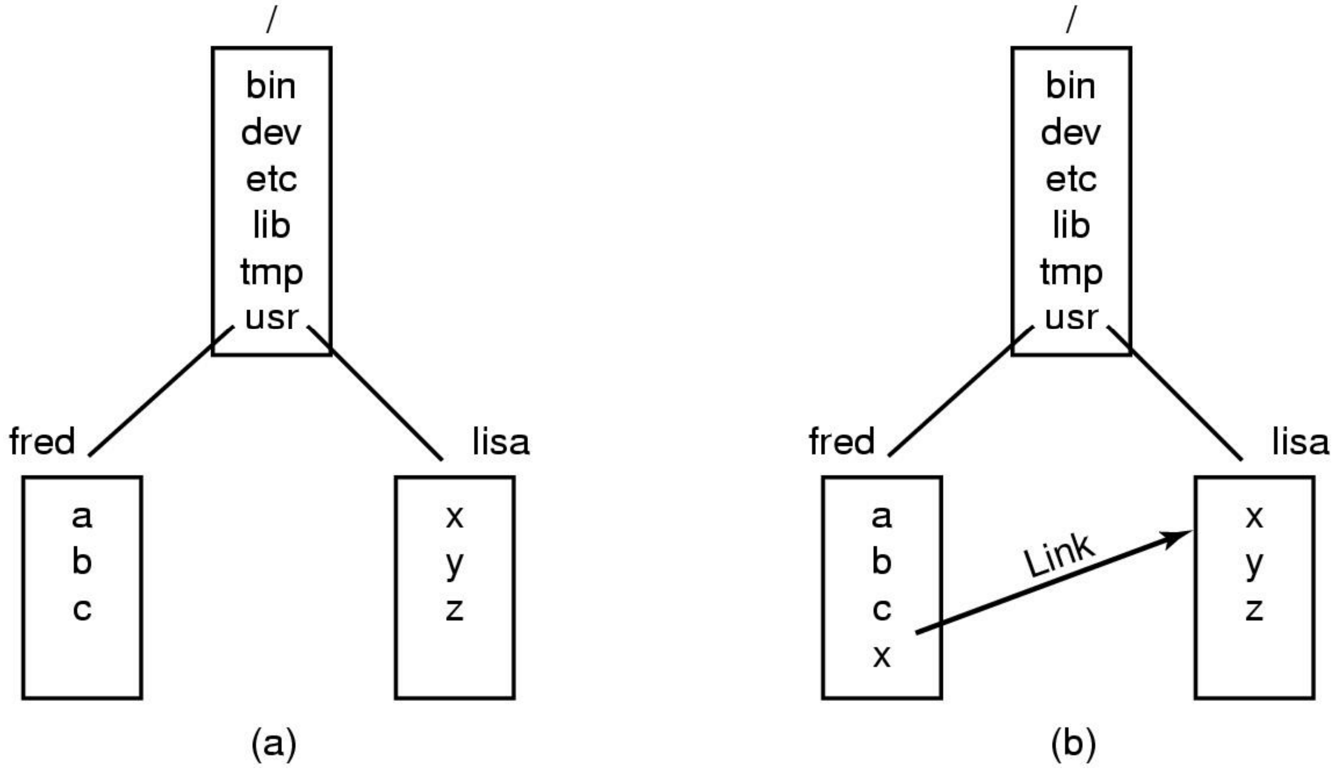
- A logical file system can consist of different physical file systems
- A file system can be mounted at any place within another file system
- When accessing the “local root” of a mounted file system, a bit in its inode identifies this directory as a so-called mount point
- Using mount respectively umount the OS manages a so called mount table supporting the resolution of path names crossing file systems
- The only file system that has to be resident is the root file system (in general on a partition of a hard disk)

Layout of a Logical Disk

- Each physical file system is placed within a logical disk partition. A physical disk may contain several logical partitions (or logical disks)
- Each partition contains space for the boot block, a super block, the inode table, and the data blocks
- Only the root partition contains a real boot block
- Border between inodes and data blocks region can be set, thus supporting better usage of the file system
  - with either few large files or
  - with many small files



Linking of Files



(a) Before linking (b) After linking

Hard Links ↔ Symbolic Links

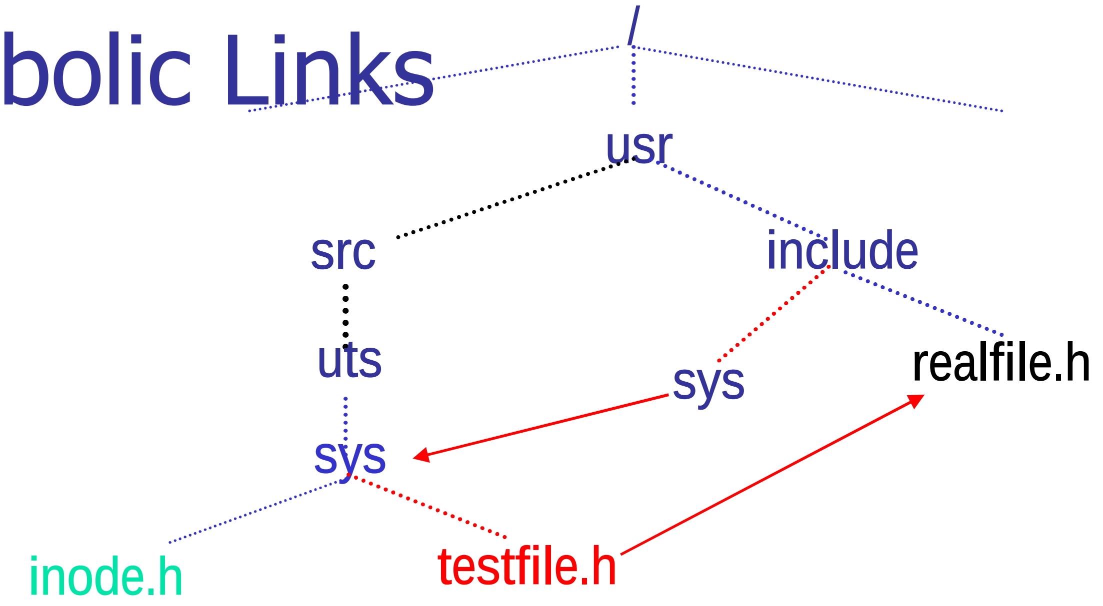
Hard link is another *file name*, i.e. ∃ another directory entry pointing to a specific file; its inode-field is the same in all hard links. Hard links are bound to the logical device (partition).

Each new hard link increases the *link counter* in file's i-node.
As long as link counter ≠ 0, file remains existing after a *rm.*
In all cases, a remove decreases link counter.

Symbolic link is a *new file* containing a pathname pointing to a file or to a directory. Symbolic links are evaluated per access. If file or directory is removed the symbolic link points to *nirwana*.

You may even specify a symbolic link to a file or to a directory currently *not present* or even currently *not existent*.

Symbolic Links



With: symlink("/usr/src/uts/sys", "/usr/include/sys/") you add a symbolic link to a directory, i.e. you create the file /usr/include/sys pointing to the directory /usr/src/uts/sys

With: symlink("/usr/include/realfile.h", "/usr/src/uts/sys/testfile") you add a file link to realfile.h

The following 3 pathnames access the same file:
/usr/include/realfile.h
/usr/include/sys/sys/testfile.h
/usr/src/uts/sys/testfile.h

Using relative path names you may benefit from hard and soft links

How to use a Symbolic Link?

What does Unix do, when accessing a symbolic link?

<u>Example of the previous slide:</u>

```
fopen(/usr/src/uts/sys/testfile.h, …);

cp(/usr/src/uts/sys/testfile.h, newfile)
```

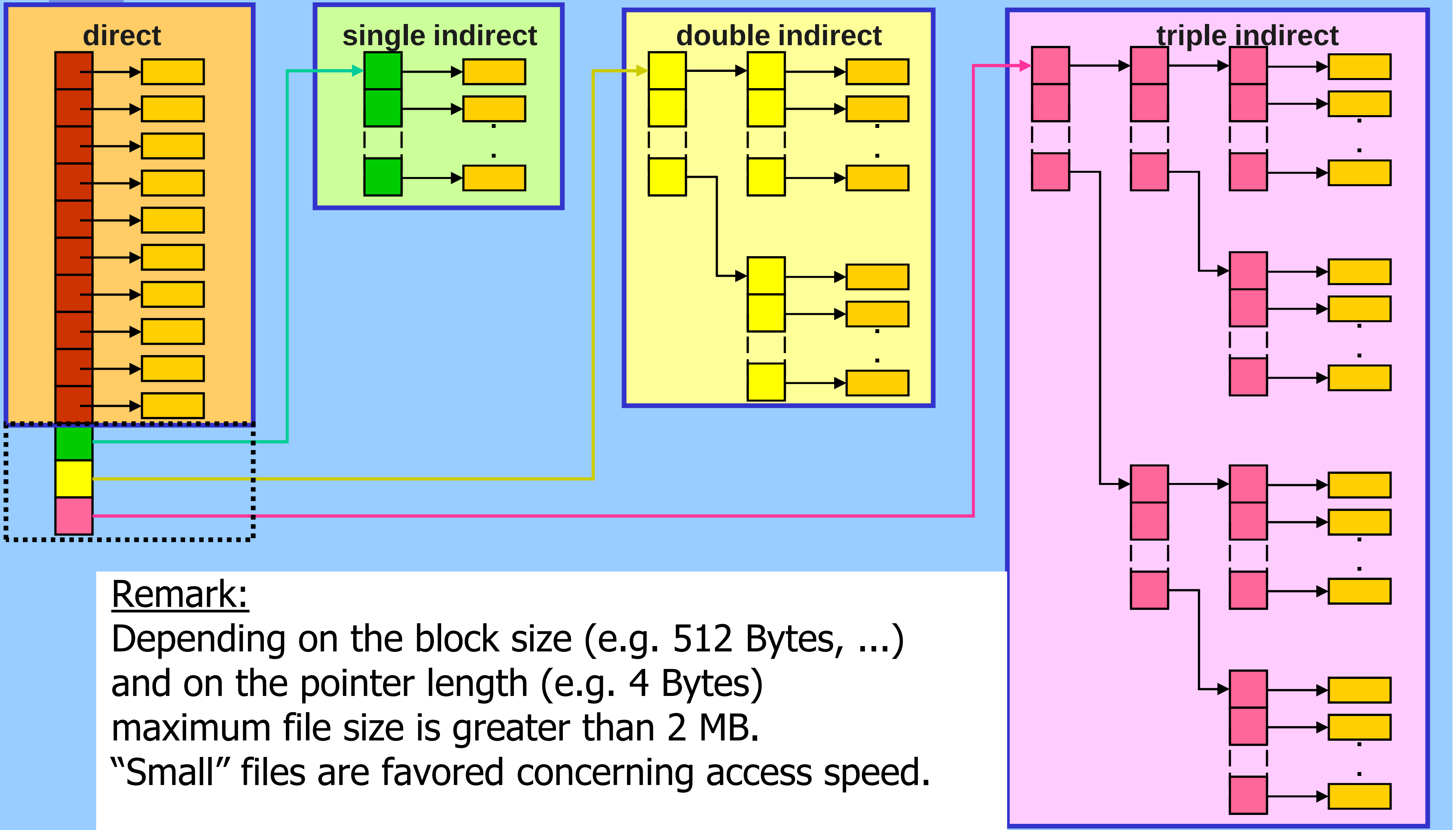
Unix File Management

- Ordinary files = array of bytes, no record structures at system level
- Types of files
  - ordinary: contents entered by user or program
  - directory: contains a list of file names
  - (including length field and inode-numbers
  - special: used to access peripheral devices
  - named: for named pipes
- Inode = file descriptor (file header) containing file attributes
  - file mode
  - link count
  - owner and group id
  - … etc.

Unix Inode

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	39	Address of first 10 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

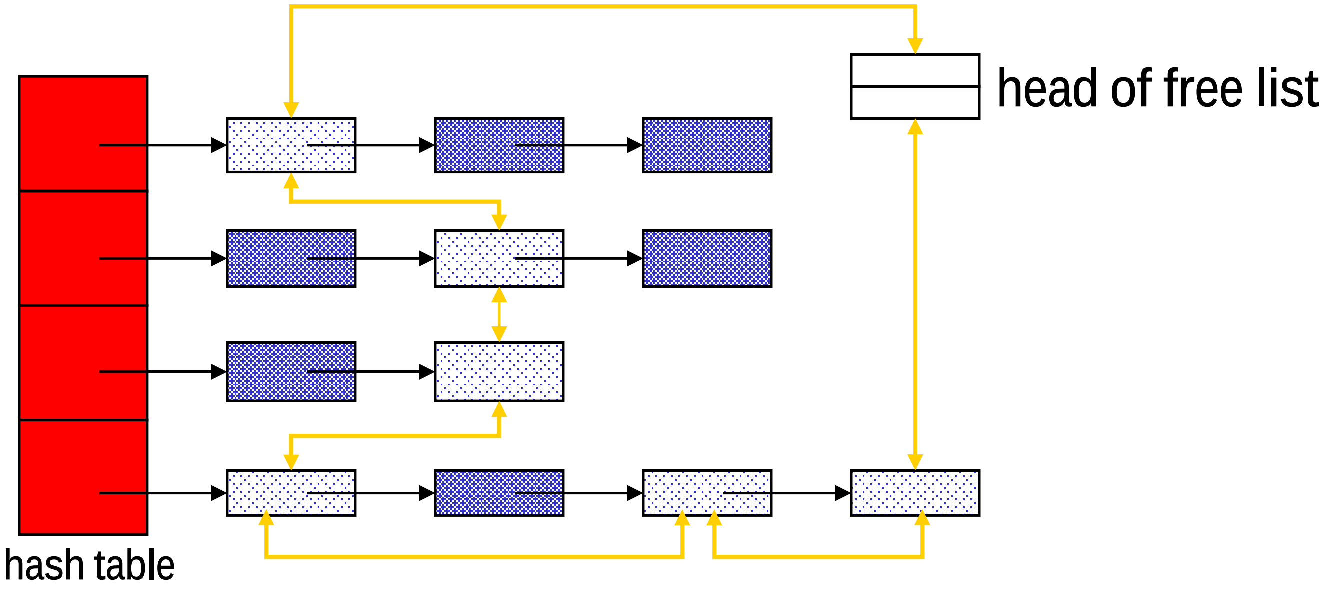
Access Structure



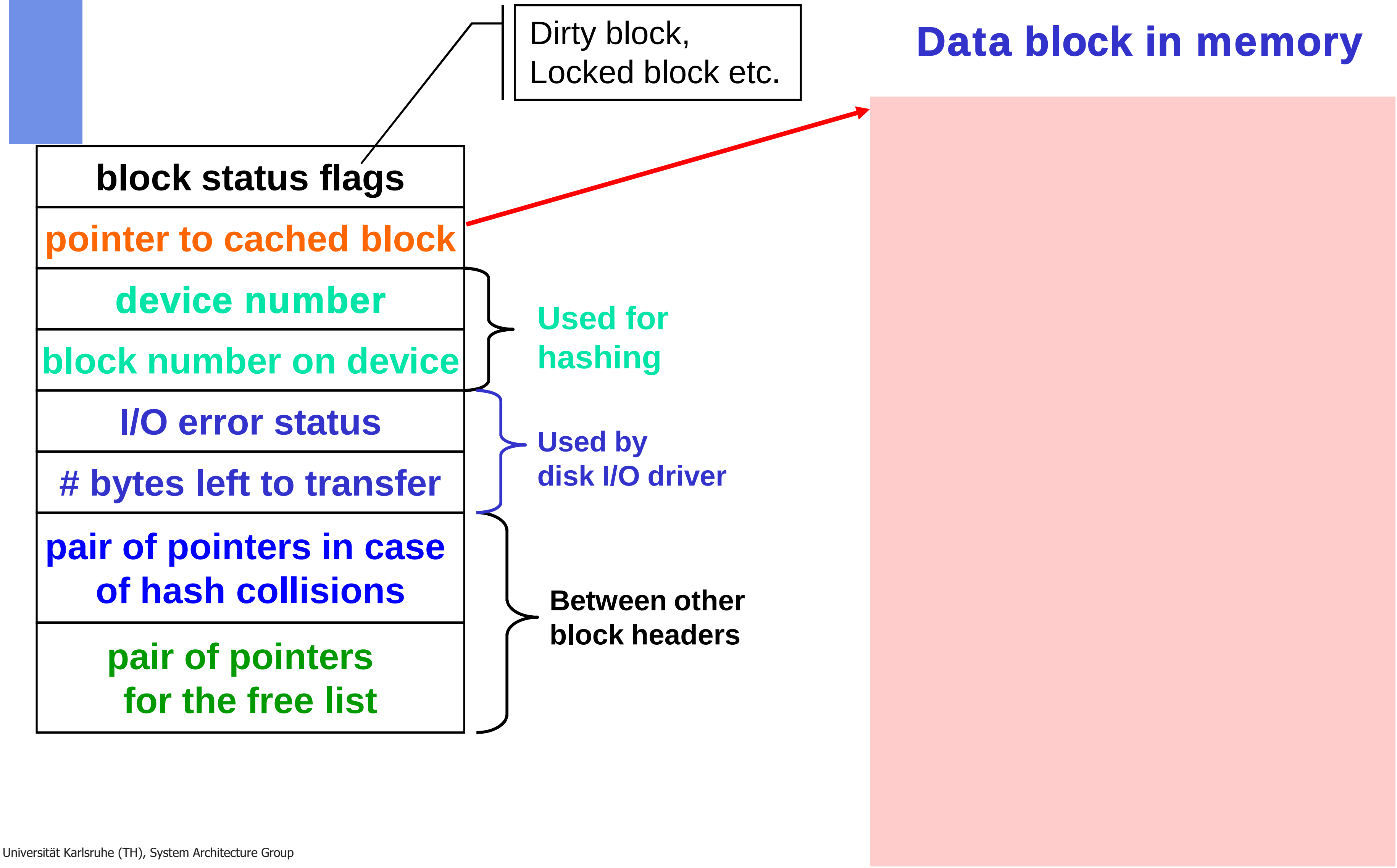
Remark:
Depending on the block size (e.g. 512 Bytes, ...)
and on the pointer length (e.g. 4 Bytes)
maximum file size is greater than 2 MB.
“Small” files are favored concerning access speed.

Buffering

- Disk blocks are buffered in main memory. Access to buffers is done via a hash table.
- Blocks with the same hash value are chained together
- Buffer replacement policy = LRU
- Free buffer management is done via a double-linked list.



UNIX Block Header



Dirty block,
Locked block etc.

Data block in memory

block status flags

pointer to cached block

device number

block number on device

Used for hashing

I/O error status

 # bytes left to transfer

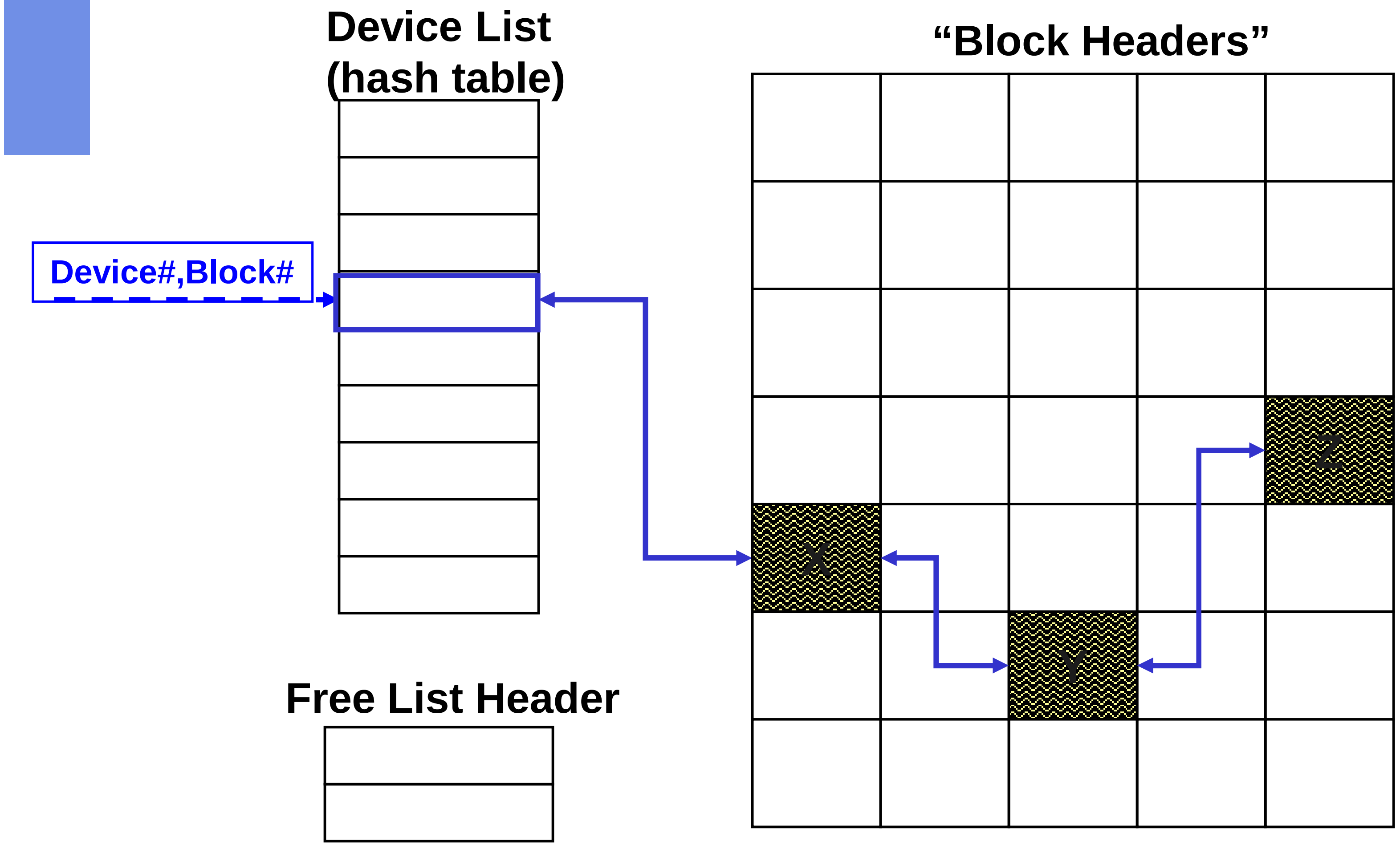
Used by disk I/O driver

pair of pointers in case of hash collisions

pair of pointers for the free list

Between other block headers

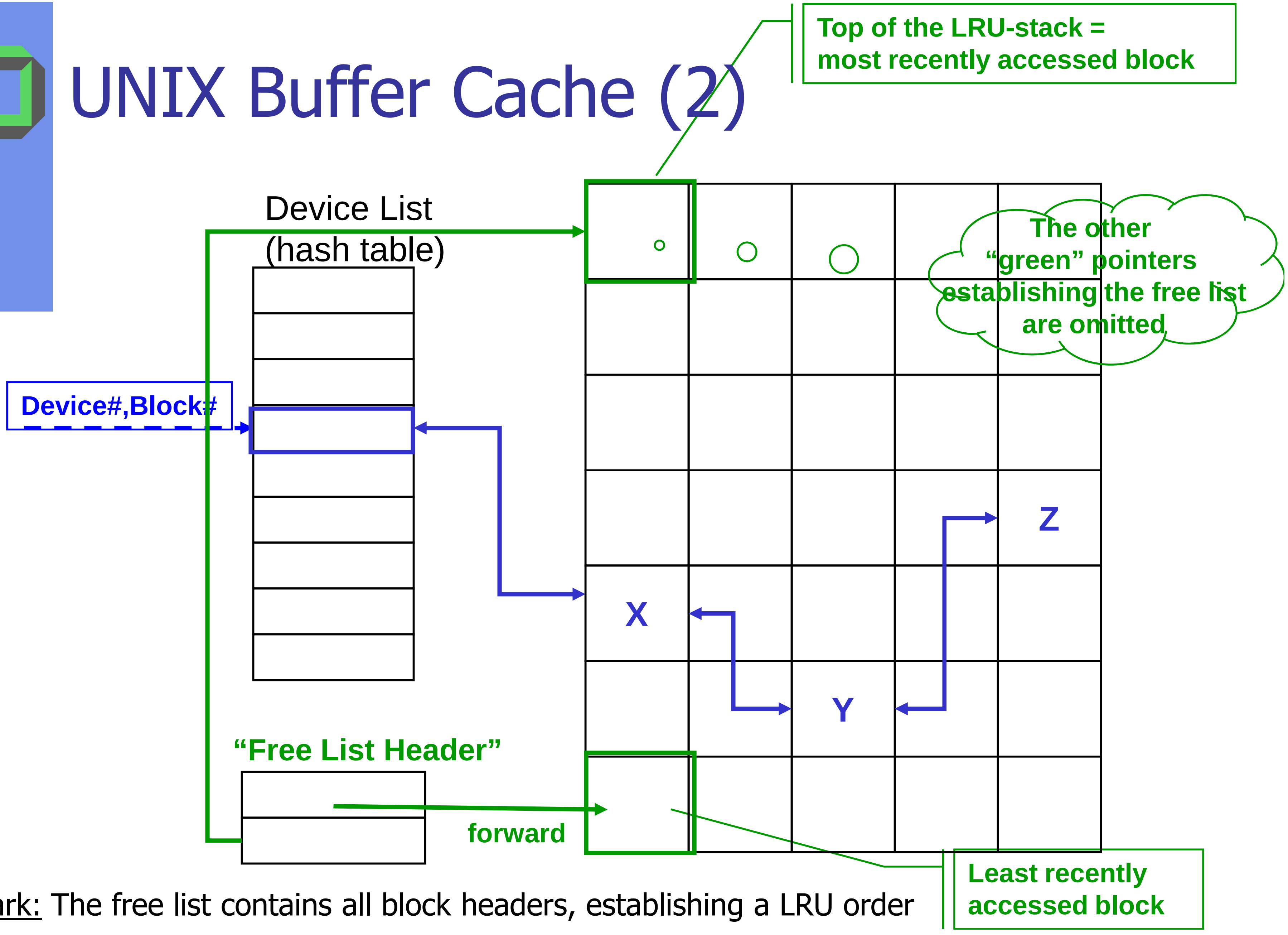
UNIX Buffer Cache (1)



Remark:

X,Y, and **Z** are block headers of blocks mapped into the same hash table entry

UNIX Buffer Cache (2)



<u>Remark:</u> The free list contains all block headers, establishing a LRU order

UNIX Buffer Cache (3)

Advantages:

- reduces disk traffic
- "well-tuned" buffer has hit rates up to 90% (according to Ousterhout 10.th SOSP 1985)
- ~ 10% of main memory for the buffer cache (recommendation for *old configurations*)

UNIX Buffer Cache (4)

Disadvantages:

- Write-behind policy might lead to
    - data losses in case of system crash and/or
    - inconsistent state of the FS

⇒ rebooting system might take some time due to fsck, i.e. *checking all directories and files* of FS

- Always *two copies* involved
    - from disk to buffer cache (in kernel space)
    - from buffer to user address space
- *FS Cache wiping* if sequentially reading a very large file from end to end and not accessing it again

Linux File System(s)

- Virtual File System VFS
  - Used to host different file systems, e.g.
    - EXT2
    - EXT3
    - Reiser FS
    - …
  - A generic interface

See: Various Linux Proseminar talks
Extract the pros & cons of the VFS approach

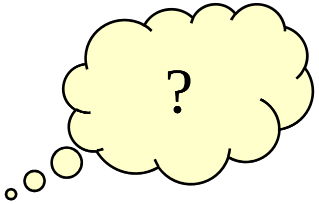
Reiser FS

Slides cover Reiser3

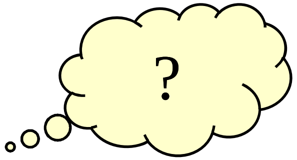
Reiser4 see:

http://www.namesys.com/v4/v4.html#repacker

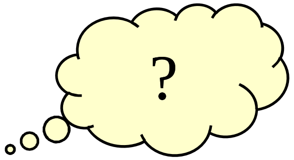
Outline



1. *What is ReiserFS?*



2. *How is it implemented?*



3. *Why did they do it like they did?*

References

Where?

- Namesys's Homepage www.namesys.com,
  - ReiserFS Architectural Overview (www.namesys.com/v4/v4.html)
  - "Future Visions" Whitepaper (www.namesys.com/whitepaper.html)
- The Source Code

http://homes.cerias.purdue.edu/~florian/reiser/reiserfs.php

- And many more papers on Reiser FS

Introduction

What?

- FS developed by Hans Reiser’s company Namesys
- First version released in mid-90s
- Part of Linux since 2.4.1 (ReiserFS V3.5)
- ReiserFS V3.6 is default FS for
    - SuSE
    - Lindows
    - Gentoo

This slides are about Reiser V3 (4)

Reiser FS Features

What?

- *fast*, but *space efficient*
- ⇒ *scalable*
  - about *10 times* faster than ext2fs
  - no penalty for small files
- *reliable*
  - journaling support
  - atomic file operations (i.e. transaction alike)
- *compatible*, but extensible
  - full support for UNIX semantics
  - sophisticated plugin system

Good design and clever implementation

Reiser FS and Disk Partitions

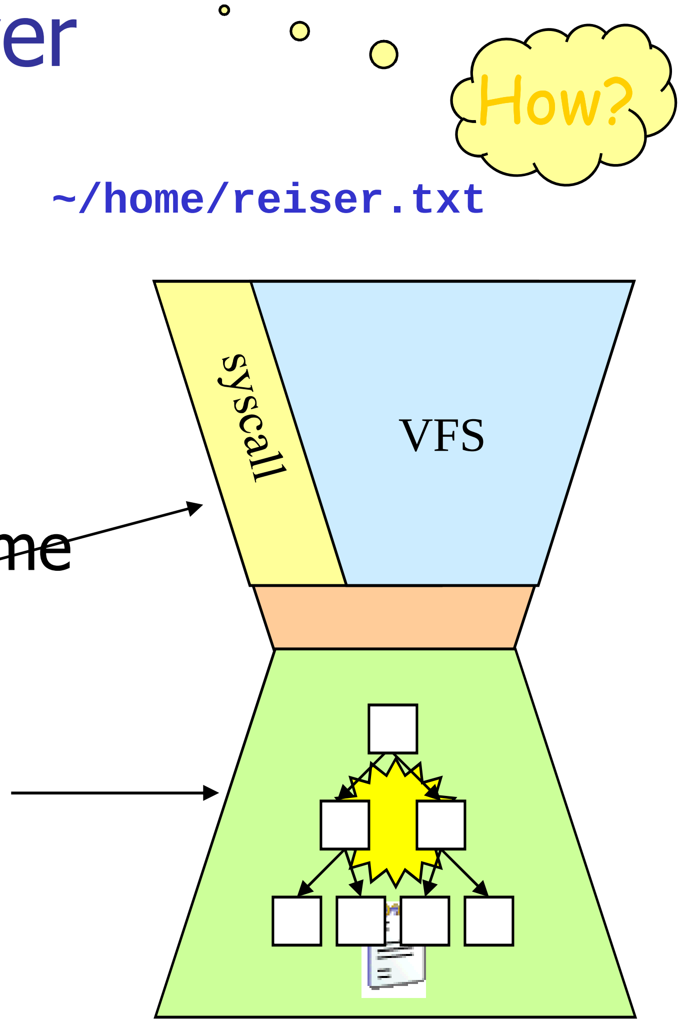
- Each ReiserFS partition might contain 2^{32} blocks
  - Partition capacity depends on size of block
- First 64 K reserved for
  - partition labels and booting info
- Next, one superblock (for complete partition)
- Next, bitmap (for partition management)
- Each *file object*→ (at least) ***one unambiguous key***
  - Directory id
  - Object id
  - Offset
  - Type

Reiser FS Features

How?

- fast, but space efficient
  - about 10 times faster than ext2fs
  - no penalty for small files
- reliable
- compatible, but extensible

Semantic vs. Storage Layer



- Theoretical point of view:

  Filesystem: Name → Object
- Two software layers:
  - *Semantic Layer:* convert name to key (mostly VFS)
  - *Storage Layer:* find object with given key
  - B+ tree based in RSF3 and *dancing tree*[1] in RSF4

[1] Try to collect neighboring data before writing to disk

Recap: B^+Trees (1)

How?

distance between leaf and root same ∀ leaves

items sorted according to keys

- Balanced *n*-way search tree

nodes have ∈ [*n/2*, *n*] children

- Tree grows only at root ⇒ stays balanced
- High fanout ⇒ flat tree: good for slow media!

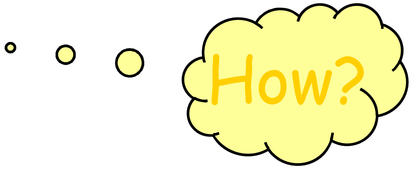
German: “Verzweigungsfaktor”

Recap: B$^+$Trees (2)

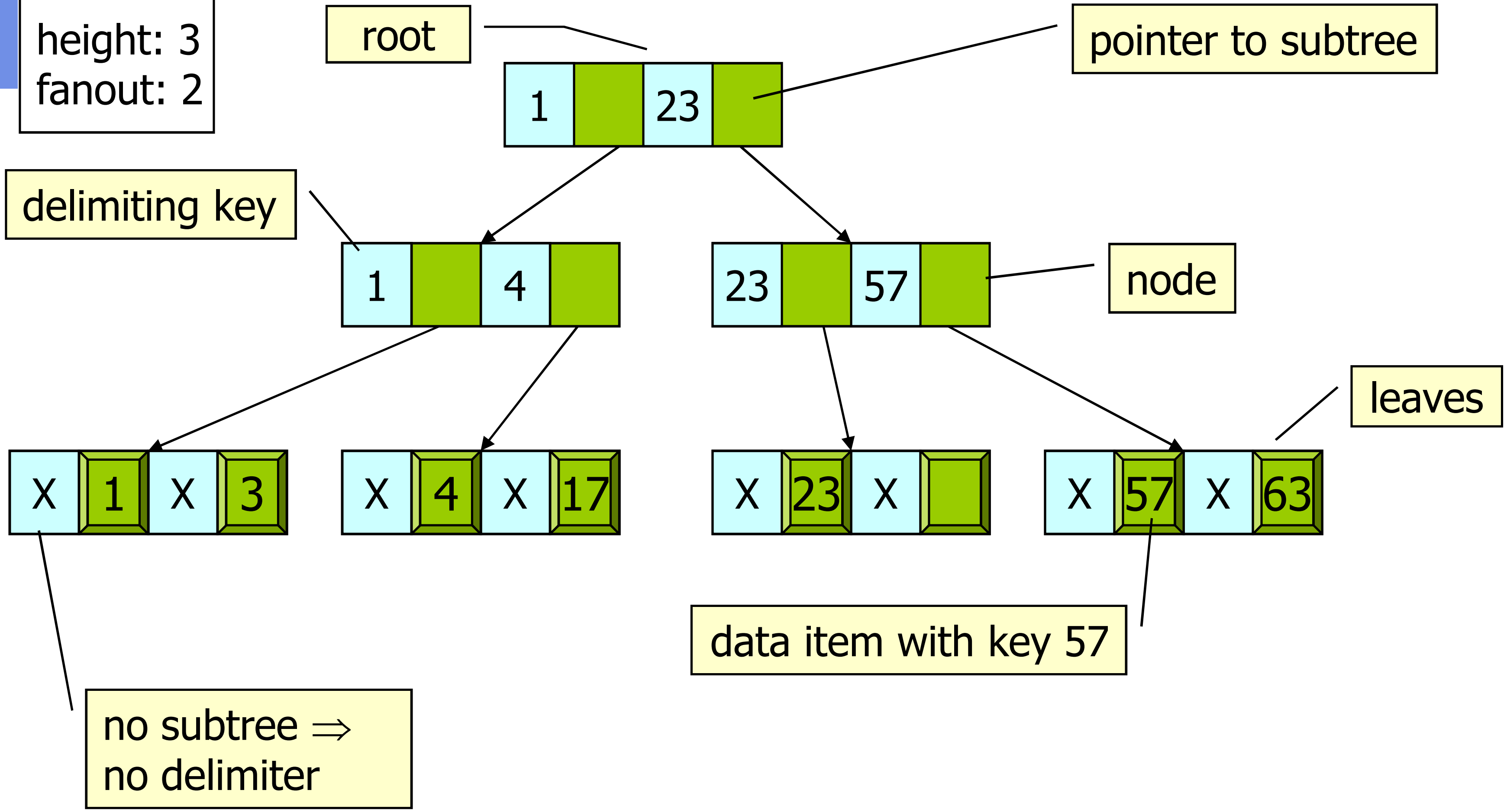
How?

- B Trees:
  - Subtrees “hang” between two data items
  - These items’ keys delimit possible keys in subtree
- B$^+$Trees:
  - Actual data items only in leaves (at lowest level) ⇒ helps caching because of increased locality
  - Roots of subtrees store delimiting *keys* (not actual items)

A Sample B$^+$Tree



height: 3
fanout: 2



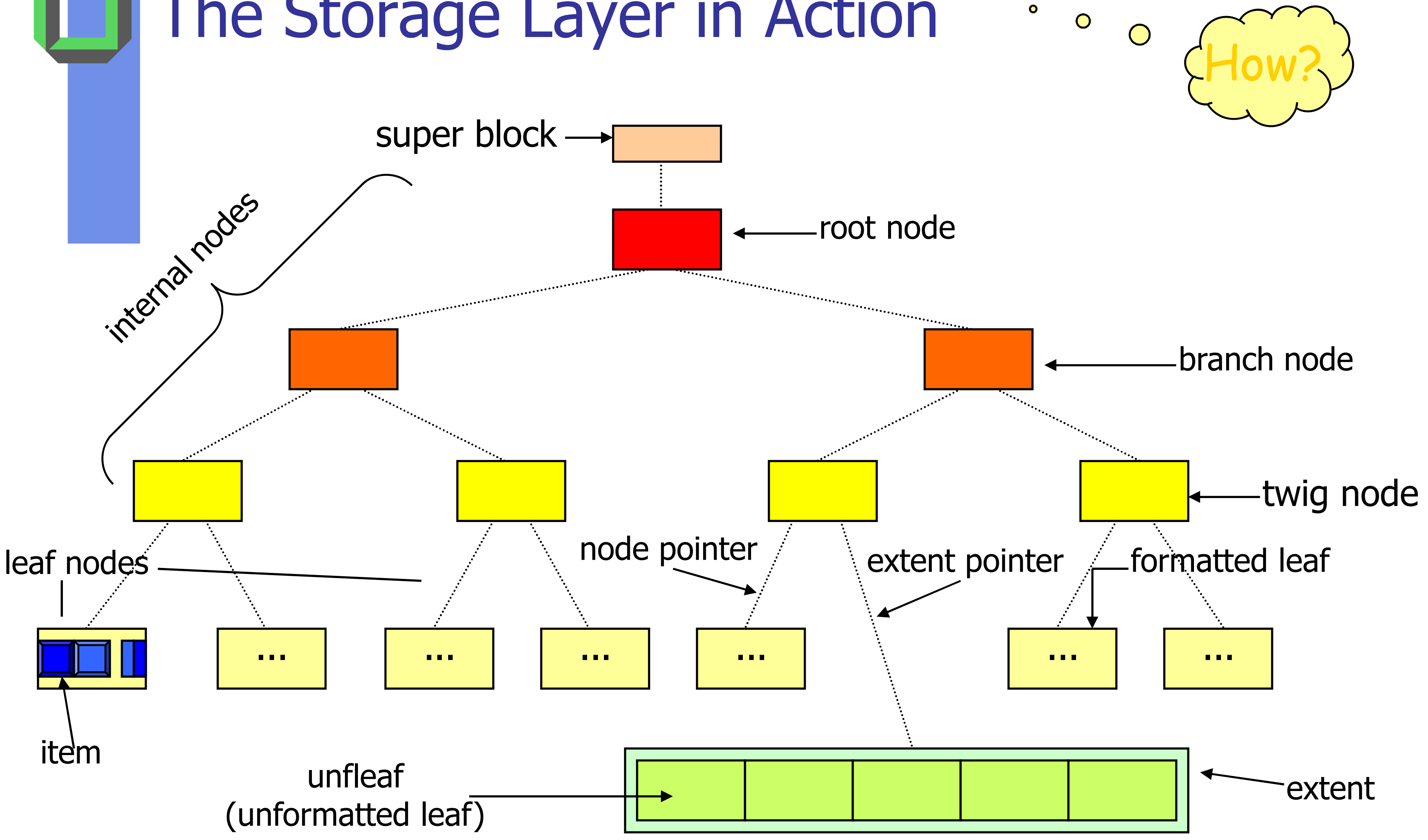
Storage Layer

How?

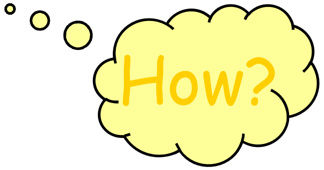
- File objects stored in B^+Tree ($2 \leq \text{height} \leq 5$)
- Node size fixed (4K) ⇒ no external fragmentation
- Object size variable: split into items that fit into nodes
  - Many small objects can be aggregated into one node (avoids internal fragmentation)
  - One large object can be distributed over multiple nodes
  - Very large objects (>16K) are stored in super-size nodes (called extents)

Additionally: some maintenance structures (superblock, allocation bitmaps ~ like ext2fs)

The Storage Layer in Action

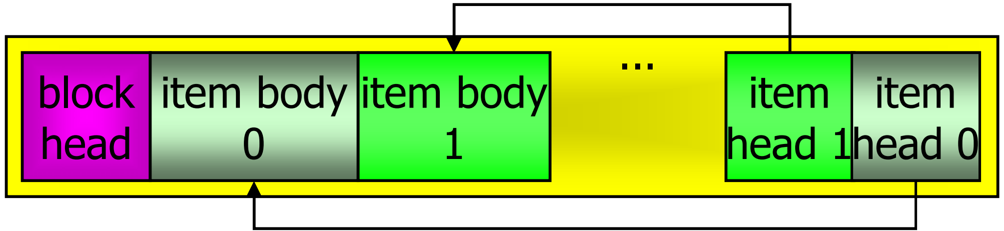


Node Layout



- unfleaf: unformatted, just raw data

- formatted leaf:



block head | item body 0 | item body 1 | … | item head 1 | item head 0

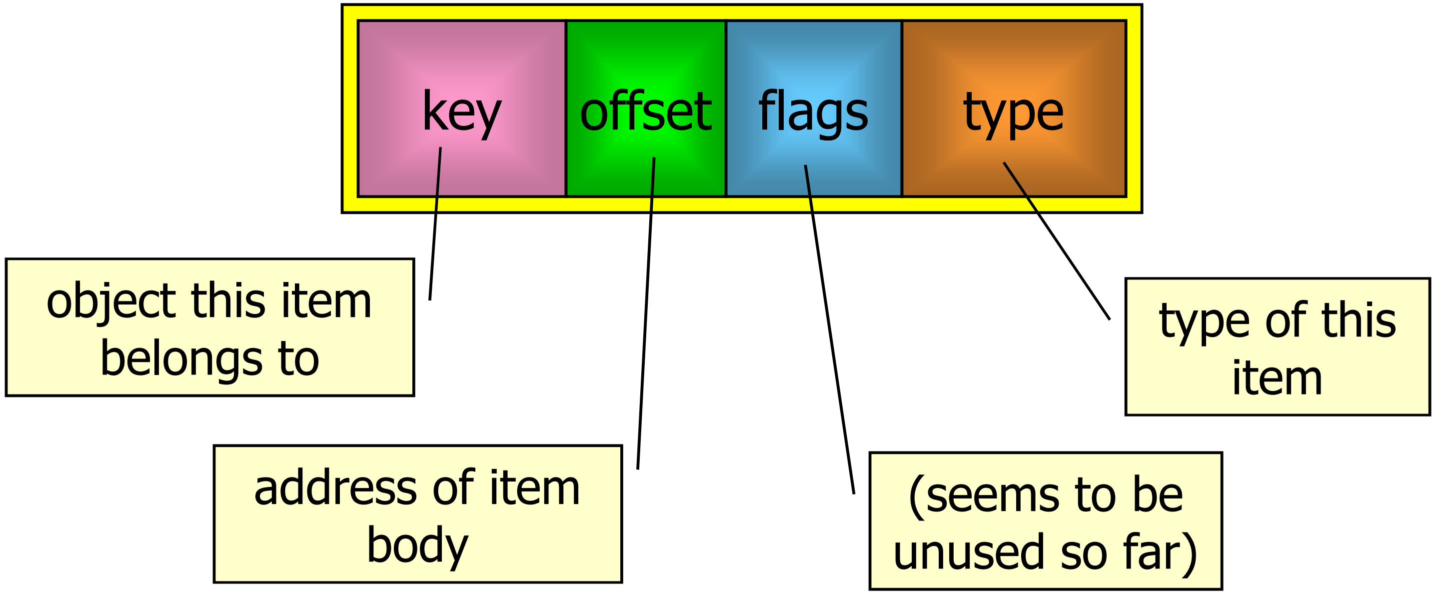
- twig node: similar structure; no actual data, only pointers

- branch node: like twig node, no extent pointers

Item Layout

How?

- item = body + head (see above)
- head layout:



Item Types

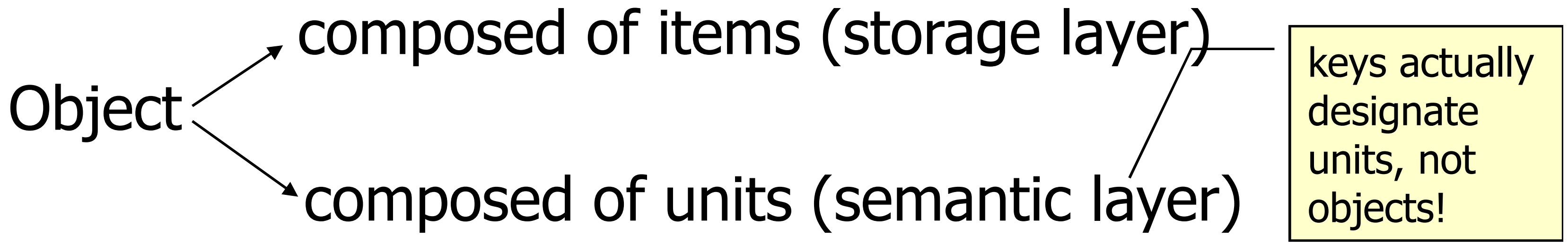
How?

- Node Pointer: points to a node (holds delimiting keys)
- Extent Pointer: points to an extent (holds extent’s size and key)
- File Body: raw data
- Stat Data: file metadata (≈ ext2fs inode)
- Directory item: hash table of file names and keys

Plugins enable to create your own item types.

Keys

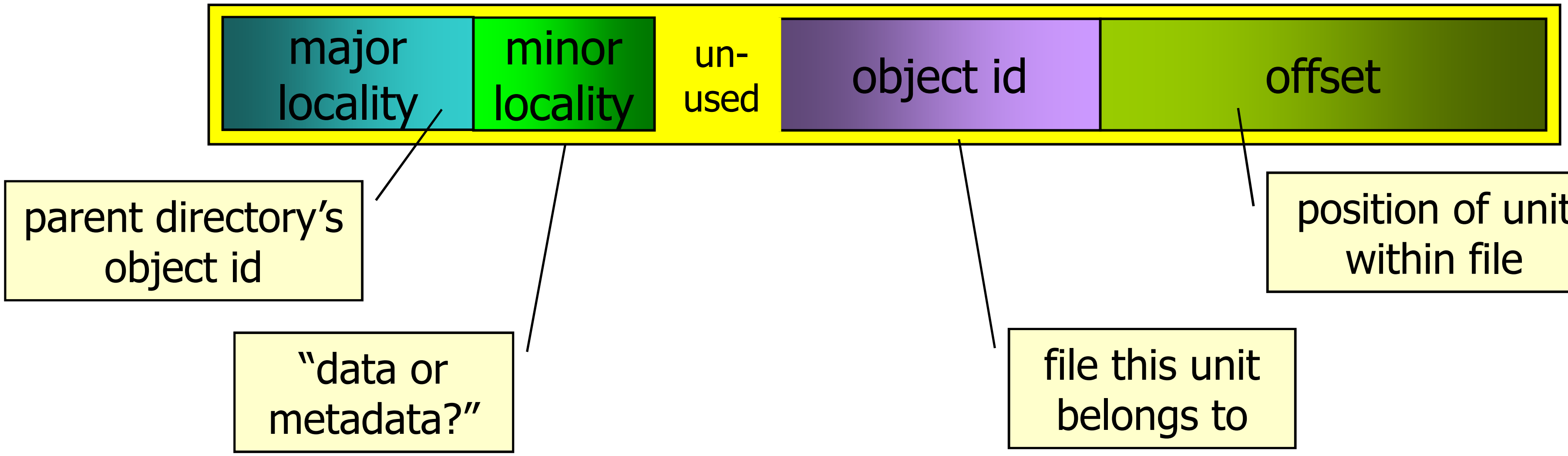
How?



- Object
  - composed of items (storage layer)
  - composed of units (semantic layer)

keys actually designate units, not objects!

- Key structure:



major locality	minor locality	un-used	object id	offset

major locality: parent directory's object id

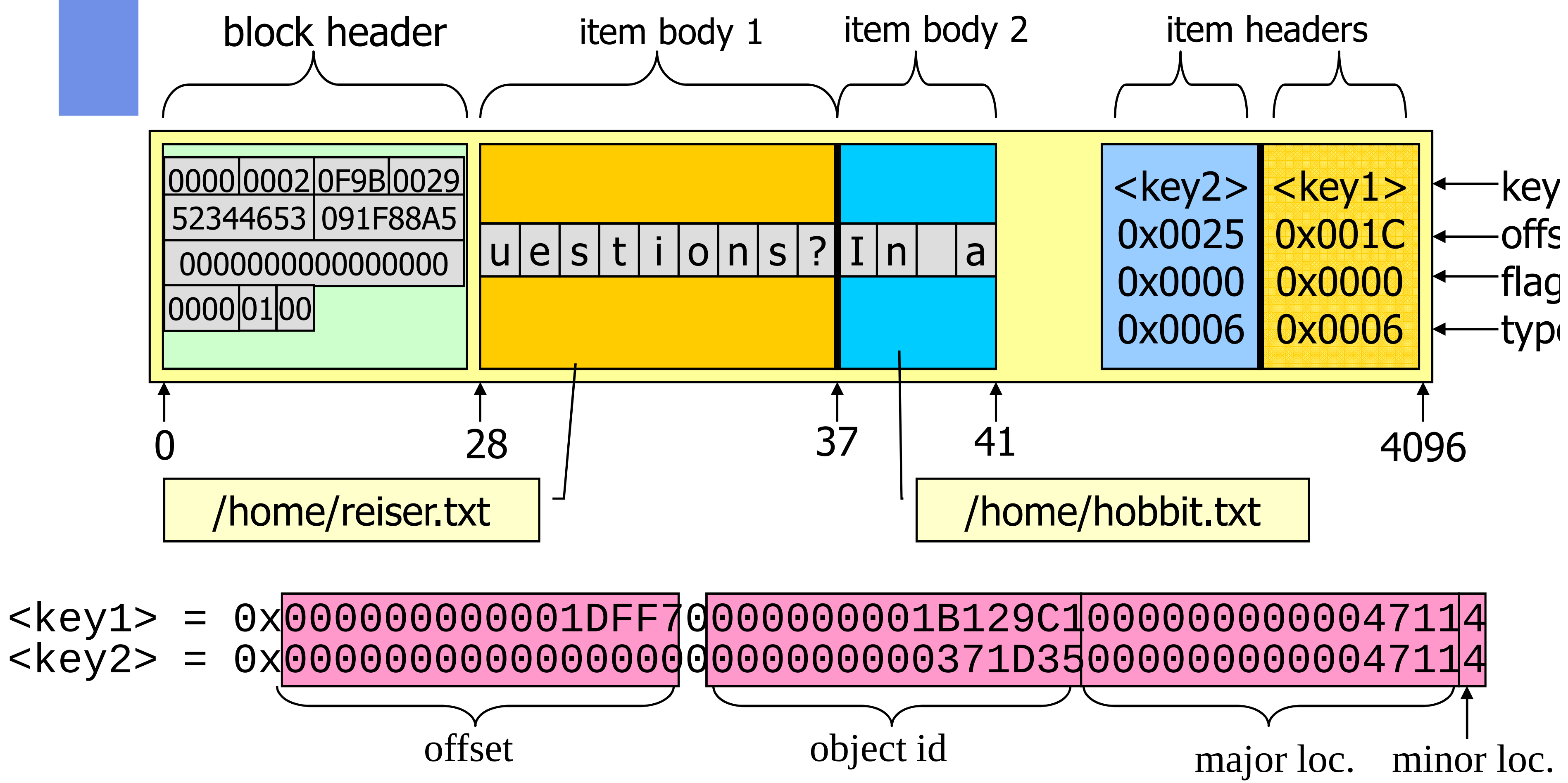
minor locality: "data or metadata?"

object id: file this unit belongs to

offset: position of unit within file

A Real-Life Leaf Node

How?



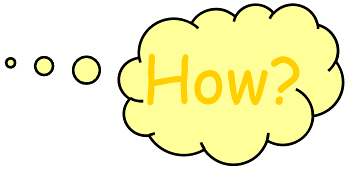
```
<key1> = 0x000000000001DFF70000000001B129C1000000000000047114
<key2> = 0x00000000000000000000000000371D35000000000000047114
```

Journaling

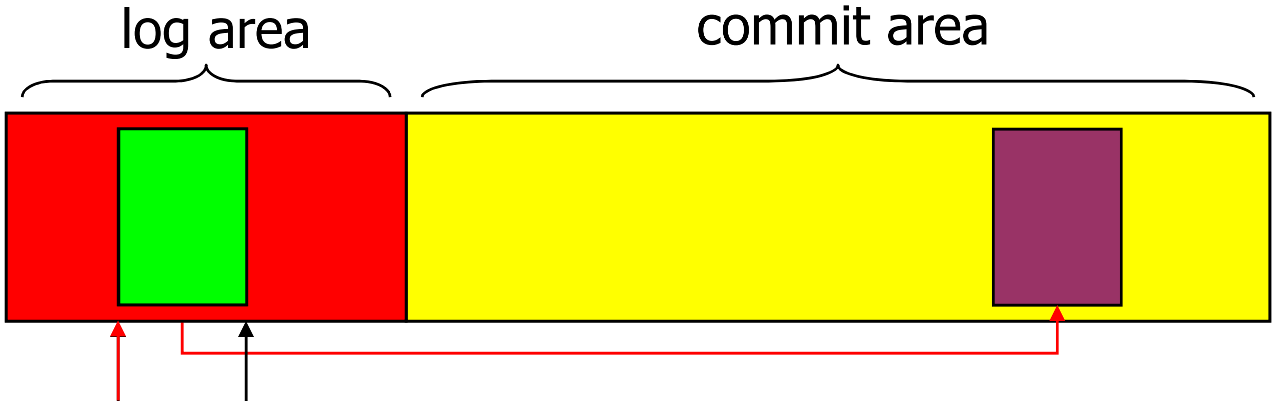
How?

- All FS modifications done as *atomic transactions*
- An atomic transaction...
  - either *completes* or does *nothing*
  - never leaves *metadata* in inconsistent state
- Intermediate state stored in a *journal/log*
- Transaction procedure:
  1. log start of transaction
  2. do transaction
  3. log completion of transaction

Fixed Logs



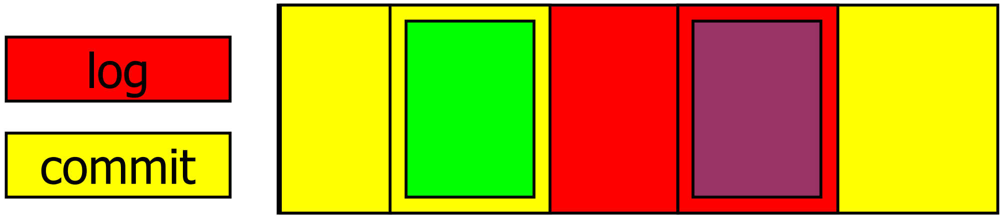
- partition disk into logging area and commit area
- copy data to logging area first, then to destination
- log structured as circular buffer ⇒ fast insertion
- but: need two copies!



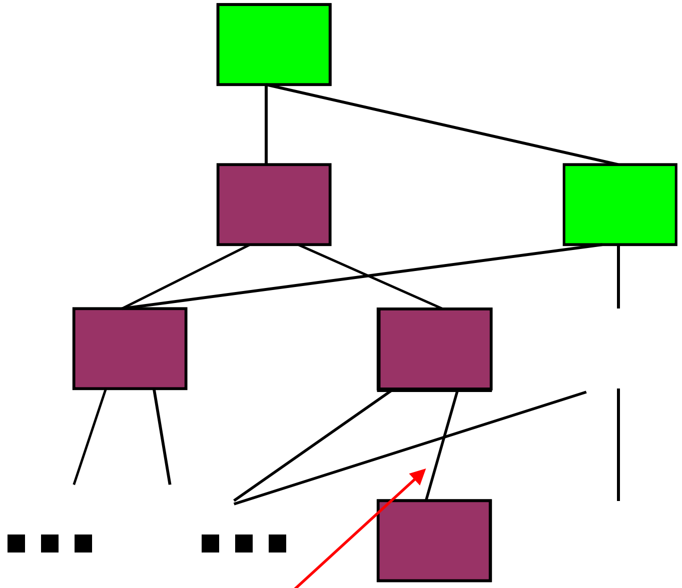
Wandering Logs

How?

- Idea: don’t move block to commit area, instead move commit area to block



but: move a node
⇒ have to move its parent node
⇒...
⇒have to move root of tree



still points to old node!

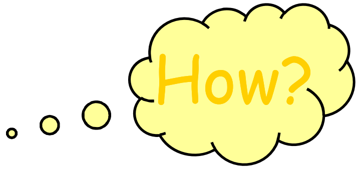
ReiserFS decides at runtime which method to use.

Log Implementation

How?

- ReiserFS partitions changed blocks into two sets
  - *Relocate Set*: blocks that can stay where they are (i.e., blocks with Wandering Log policy)
  - *Overwrite Set*: blocks that have to be copied somewhere else (i.e., blocks with Write Twice policy)
- Transaction represented by its Wander List: records destinations of blocks from Overwrite Set
- Wander Lists of incomplete transactions recorded at fixed disk location
- If system crashes: Wander List "replayed" on next boot

Plugins



- ReiserFS can be customized through plugins:
  - semantic layer
    - **object** plugins: provide object semantics (standard: regular file, symlink, directory)
    - **security** plugins: control access to object data (used by object plugins)
  - storage layer
    - **hash** plugins: implement hash functions to order directory entries
    - **tail policy** plugins: decide whether an object should be split into items or put into an extent
    - **item** plugins: define new kinds of items
    - **node** plugins: define new kinds of nodes
  - plus some more esoteric ones...
- Reiser4: plugins static, dynamic ones to come

Reiser FS and VFS

How?

- VFS requests: handed to object plugins
- plugin: knows how to handle them (e.g., standard file plugin supports traditional UNIX semantics)
- system call `reiser4`: access to more advanced ReiserFS features (e.g., fancy plugins)
- but: using a separate system call just for ReiserFS is tricky and overall just

👎 DROP DEAD UGLY 👎

(Reiser promises it is only a temporary solution ...)

Reiser FS Features

How?

- ✓ fast, but space efficient
  - about 10 times faster than ext2fs
  - no penalty for small files
- ✓ reliable
  - journaling support
- ✓ compatible, but extensible
  - full support for UNIX semantics
  - sophisticated plugin system

The Big Picture

Why?

- Speed and reliability: hardly need justification
- Efficient small file handling
    - more important than you might realize: Max’s home installation has more than 80% files <16K!
    - Allows nice generalization: file attributes can be stored as separate, small files
    - Whole databases could be integrated into the file system
- Future versions:
    - Reiser5: distributed FS
    - Reiser6: non-hierarchical file lookup (DB queries)

Ultimate Goal:

Unifying FS, database and search engine.

Evaluation

So?

- Pro:
  - + speed
  - + reliability
  - + extensibility
- Con:
  - difficult setup (no easy conversion from ext2fs)
  - Linux only
  - complex code:

```
ext2fs:   4659 LOC
ext3fs:   7840 LOC
Reiser3: 20780 LOC
Reiser4: 92061 LOC
```

cf. `fs/ext2fs/super.c:174:`

```
static void init_once(void * foo, …)
```

but: relatively clean code; no `foo`s

Summary

- Reiser FS is a modern file system supporting
  - fast and efficient file handling through tree structured storage layer
  - transaction safety and easy crash recovery through journaling
  - extensibility and scalability through plugins
- Its more advanced semantics, however, cannot adequately be supported by commodity OSes like Linux.