VFS

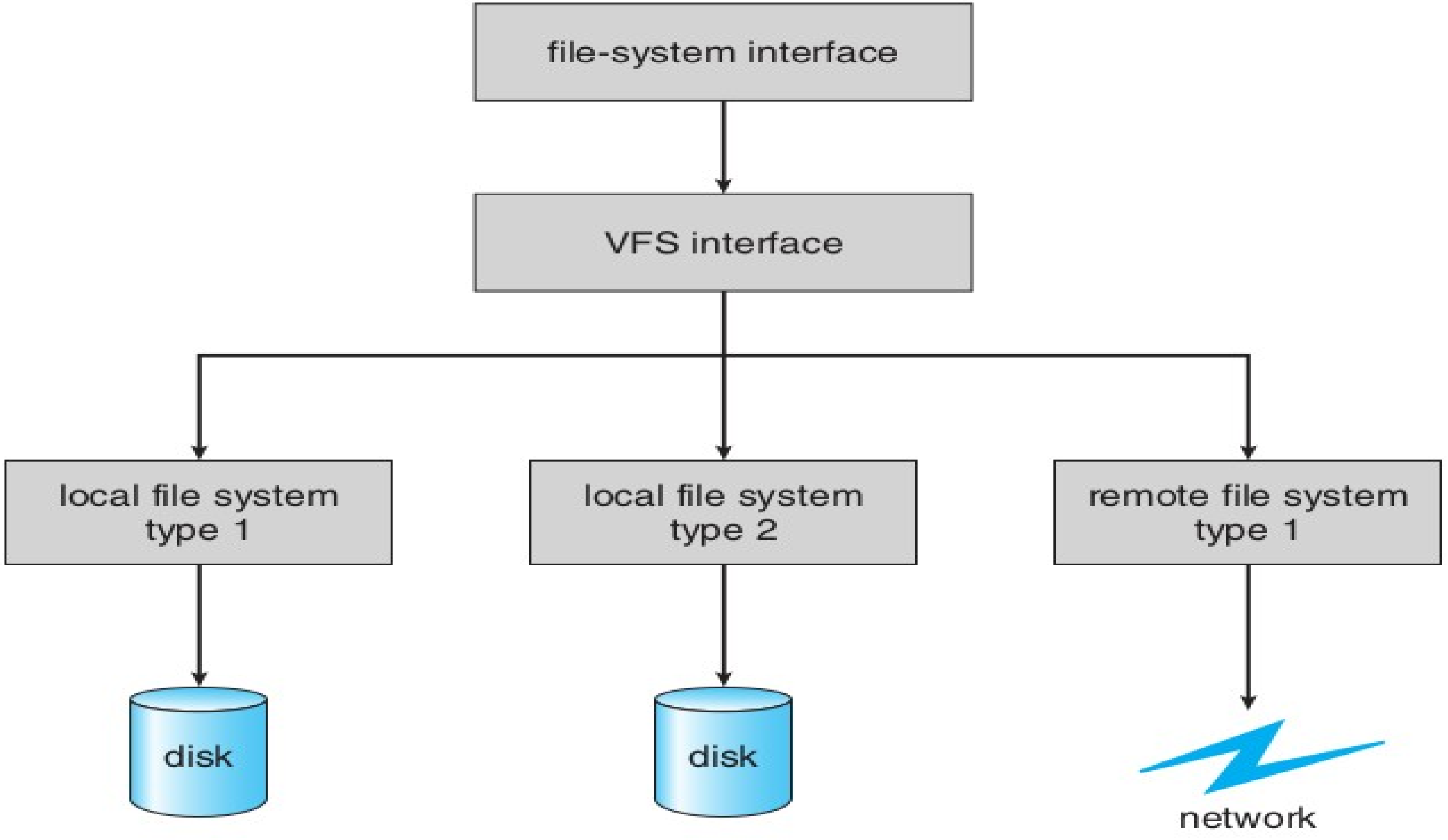


Figure 15.5 Schematic view of a virtual file system.

VFS

- Consider this
  - */dev/*sda1 is “/”
  - */dev/*sda2 is mounted on “/a/b” folder
  - How does this work in kernel?
    - open(“/a/b/c/d”, O_RDONLY)
- Consider xv6 code
  - sys_open -> namei -> namex -> (skipelem, dirlookup, ilock)
  - Dirlookup() of “c” in “/a/b” should return : Not the inode of “c” on */dev/*sda1 but inode of “/” on /dev/sda2

VFS

- Object Oriented Programming in C (let's see example of this)
  - Clever use of function pointers
- There is an "abstract" file system class (VFS), and there are concrete file system classes (ext2, vfat, ...)
  - sys_read → fileread → readi() becomes
  - sys_read → fileread → (i->-i_ops->read)()
- Inode is a generic inode
  - Contains file system specific inode pointer
  - And file system specific inode operations
  - Fields setup during namei()

```
struct inode_operations {
        int (*readi) (int, char *, int);
        int (*writei) (int, char *, int);
        ....
}
struct inode {
     int mode,
     Int uid;
      ....
     void *inode_specific;
     struct inode_ops i_ops;
}
```

Efficiency and Performance
(and the risks created
while trying to achieve it!)

Efficiency

- Efficiency dependent on:
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures

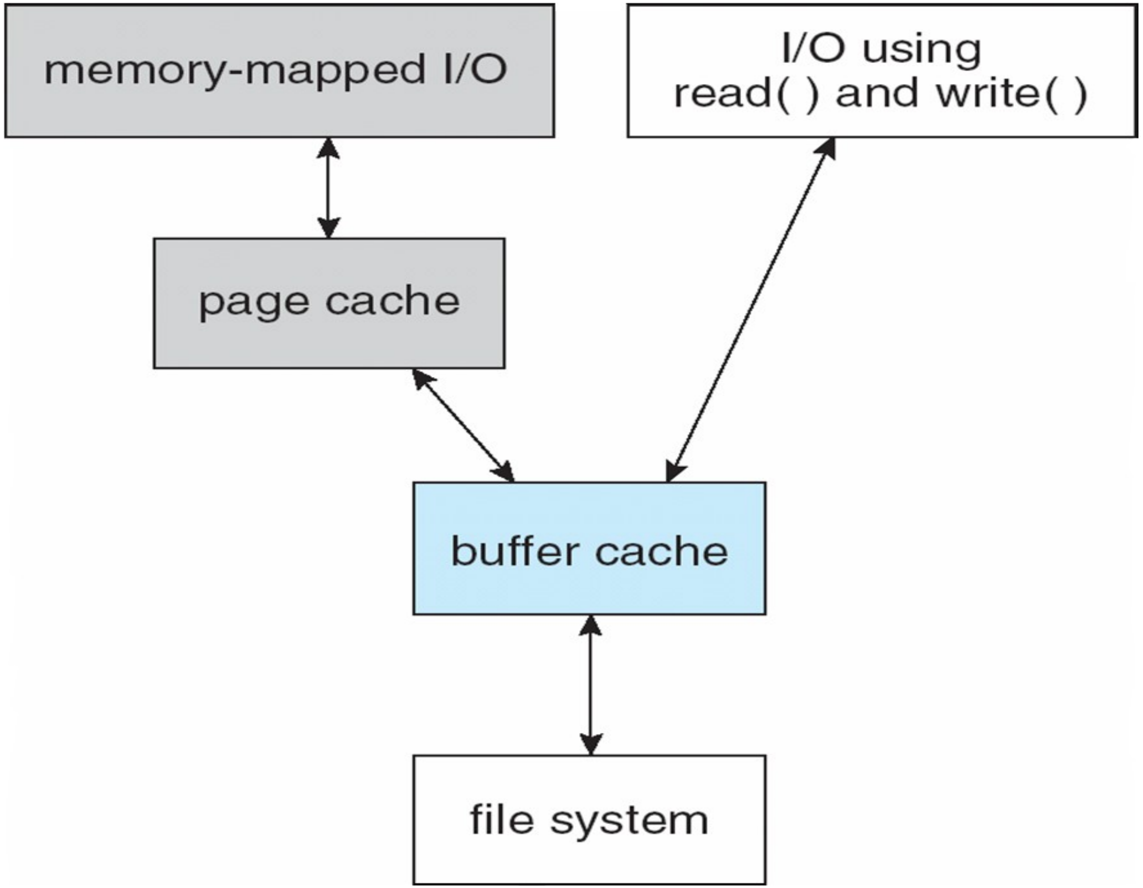
Performance

- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement
- Asynchronous writes more common, buffer-able, faster
- Free-behind and read-ahead – techniques to optimize sequential access
- Reads frequently slower than writes

Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

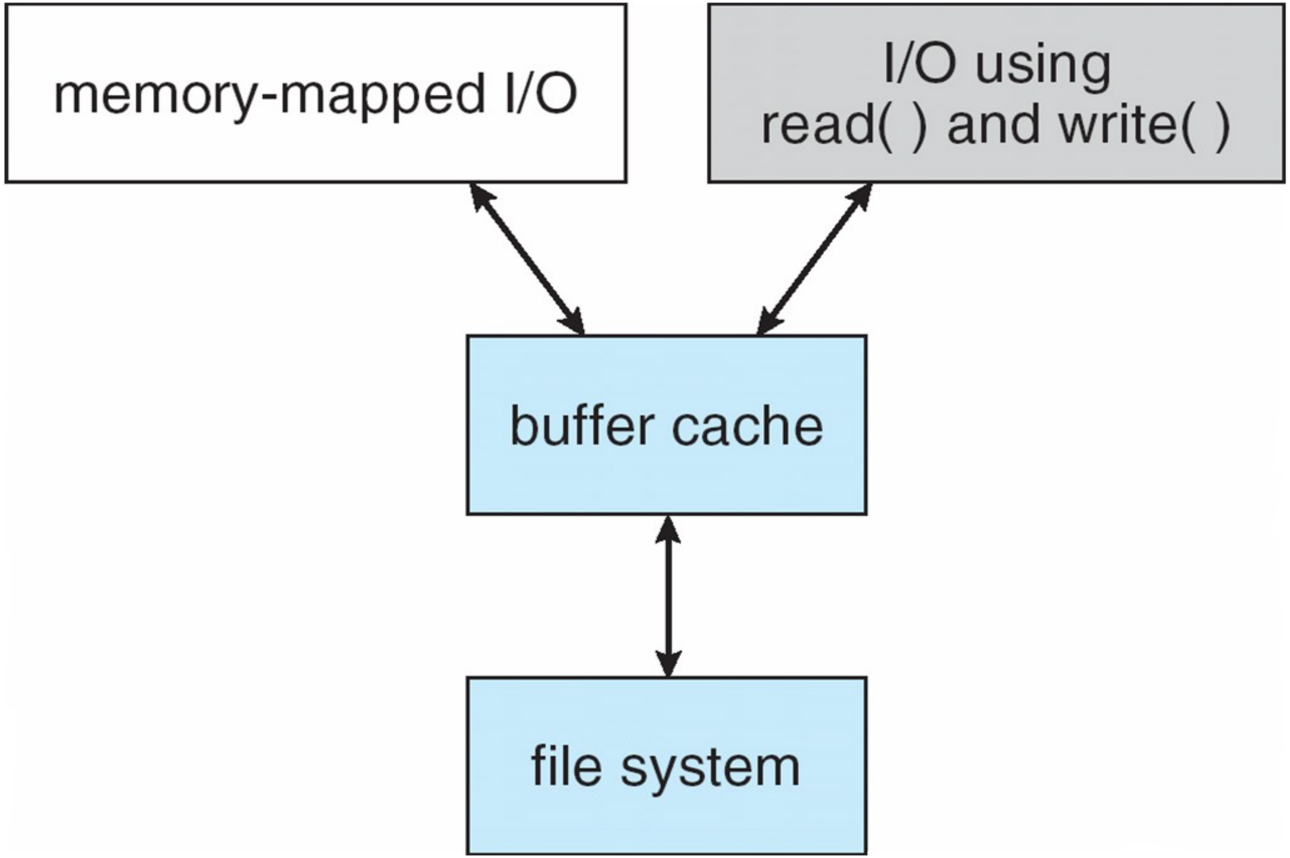
I/O Without a Unified Buffer Cache



Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache



Recovery

- Problem. Consider creating a file on ext2 file system.
  - Following on disk data structures will/may get modified
  - Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
  - All cached in memory by OS
- Delayed write – OS writes changes in its in-memory data structures, and schedules writes to disk when convenient
  - Possible that some of the above changes are written, but some are not
  - Inconsistent data structure! --> Example: inode table written, inode bitmap written, but directory data block not written

Recovery

- fsck: Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup
- Faster recovery? - “log structured file system” or “journaling file system” can help

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log (sequentially)
  - Sometimes to a separate device or section of disk
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system structures
  - When the file system structures are modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

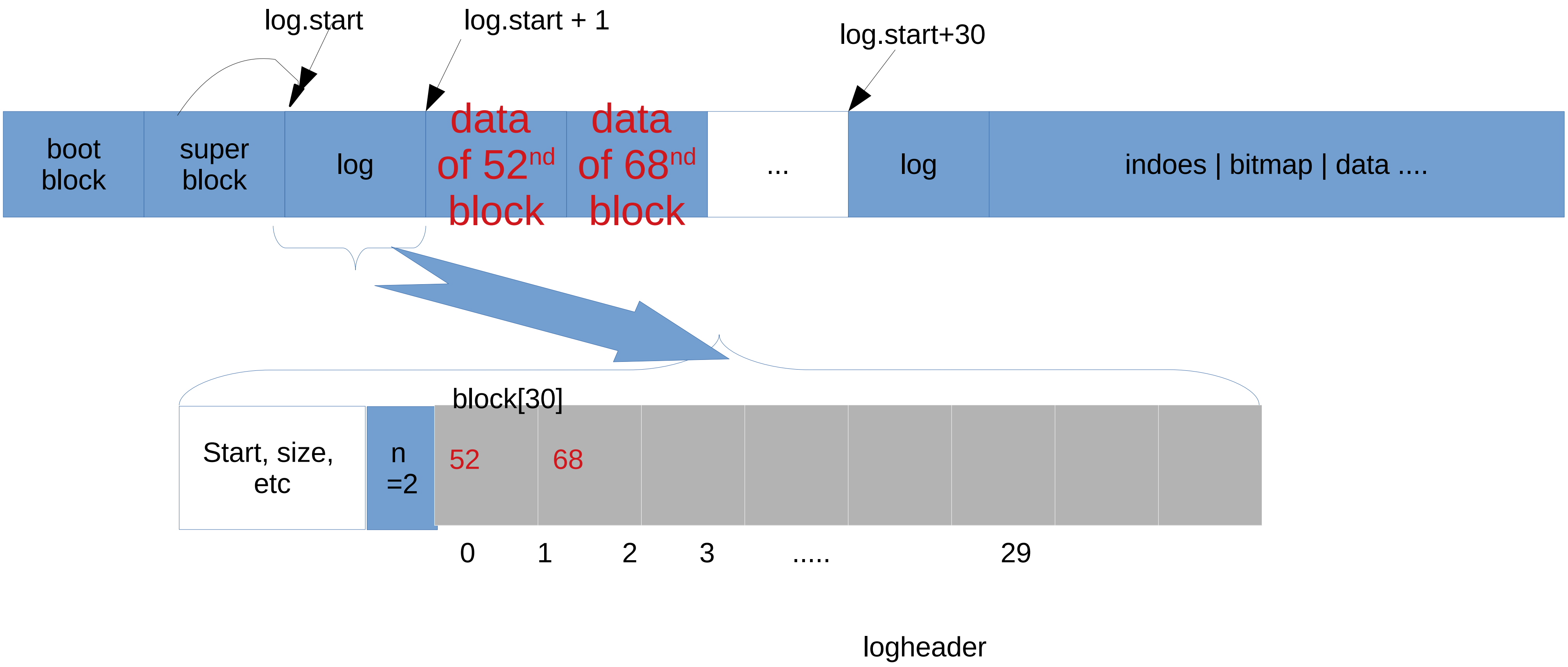
Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!

log in xv6

- a mechanism of recovery from disk
- Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)
  - some writes succed and some don’t
  - leading to inconsistencies on disk
- In the log, all changes for a ‘transaction’ (an operation) are either written completely or not at all
- During recovery, completed operations can be “rerun” and incomplete operations neglected

log on disk


log.start
log.start + 1
log.start+30
boot block
super block
log
data of 52nd block
data of 68nd block
...
log
indoes | bitmap | data ....
block[30]
Start, size, etc
n =2
52
68
0
1
2
3
.....
29
logheader

log in xv6

- xv6 system call does not directly write the on-disk file system data structures.
- A system call calls begin_op() at begining and end_op() at end
  - begin_op() increments log.outstanding
  - end_op() decrements log.outstanding, and if it’s 0, then calls commit()
- During the code of system call, whenever a buffer is modified, (and done with)
  - log_write() is called
  - This copies the block in an array of blocks inside log, the block is not written in it’s actual place in FS as of now
- when finally commit() is called, all modified blocks are copied to disk in the file system

log

```
struct logheader { // ON DISK
  int n; // number of entries in use in block[] below
  int block[LOGSIZE]; // List of block numbers stored
};
struct log { // only in memory
  struct spinlock lock;
  int start;  // first log block on disk (starts with logheader)
  int size; // total number of log blocks (in use out of 30)
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.
  int dev; // FS device
  struct logheader lh;  // copy of the on disk logheader
};
struct log log;
```

Typical use case of logging

```
/* In a system call code * /
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

prepare for logging. Wait if logging system is not ready or ‘committing’. ++outstanding

read and get access to a data block – as a buffer

modify buffer

note down this buffer for writing, in log. proxy for bwrite(). Mark B_DIRTY. Absorb multiple writes into one.

Syscall done. write log and all blocks. --outstanding.

If outstanding = 0, commit().

match colors in code and comments on right-side

Example of calls to logging

```
//file_write() code
begin_op();
ilock(f->ip);
   /*loop */ r = writei(f->ip, ...);
iunlock(f->ip);
end_op();
```

- each writei() in turn calls bread(), log_write() and brelse()
  - also calles iupdate(ip) which also calls bread, log_write and brelse
- Multiple writes are combined between begin_op() and end_op()

Logging functions

- Initlog()
  - Set fields in global log.xyz variables, using FS superblock
  - Recovery if needed
  - Called from first forkret()
- Following three called by FS code
- begin_op(void)
  - Increment log.outstanding
- end_op(void)
  - Decrement log.oustanding and call commit() if it's zero
- log_write(buf *)
  - Remember the specified block number in log.lh.block[] array
  - Set the block to be dirty

- write_log(void)
  - Called only from commit()
  - Use block numbers specified in log.lh.block and copy those blocks from memory to log-blocks
- commit(void)
  - Called only from end_op()
  - write_log()
  - Write header to disk log-header
  - Copy from log blocks to actual FS blocks
  - Reset and write log header again