

Persistence and Consistency in File Systems

COMP 321

Dave Johnson



COMP 321

Copyright © 2026 David B. Johnson

Page 1

1

Appearances vs. Performance Reality

Kernel reading and writing files appears to be synchronous and unbuffered

- Immediately after return from a read() call, the data is available
- Immediately after a write() call, the process may freely reuse the user buffer

But, really, the kernel buffers file data in a cache inside the kernel

- Data written on a write() is immediately copied into the cache but may be delayed in writing to the disk
 - e.g., combine multiple changes to the same block into a single disk write
- A read() causes the file block to be read into the cache
 - e.g., multiple future read() or write() can just access the cached block
- ***Greatly improves performance by reducing disk I/O (and its delays)***

COMP 321

Copyright © 2026 David B. Johnson

Page 2

2

A Simple Example

Consider the following simple program

- Without caching in the kernel, this program does
 - 40,960 disk writes
 - 40,950 disk reads to get the block to add the next char to
 - **Elapsed time 5.84 seconds!**
- With caching in the kernel, this program does
 - 10 disk writes
 - 0 disk reads
 - **Elapsed time 0.04 seconds!**

```
int main(void)
{
    int fd;
    int i;
    char ch = 'X';

    fd = creat("data.txt", 0666);

    for (i = 0; i < 10 * 4096; i++)
        write(fd, &ch, 1);

    exit(0);
}
```

3

A Few Questions

- When is data from a write() actually persistent on the disk?
- What can we rely on for ordering of different writes to disk?
- Does a read() always return the actual up-to-date data that is on disk?
- What about changes to a file's inode (e.g., recording the block numbers of newly added file blocks)?
- What about for file systems accessed over a network rather than local?

4

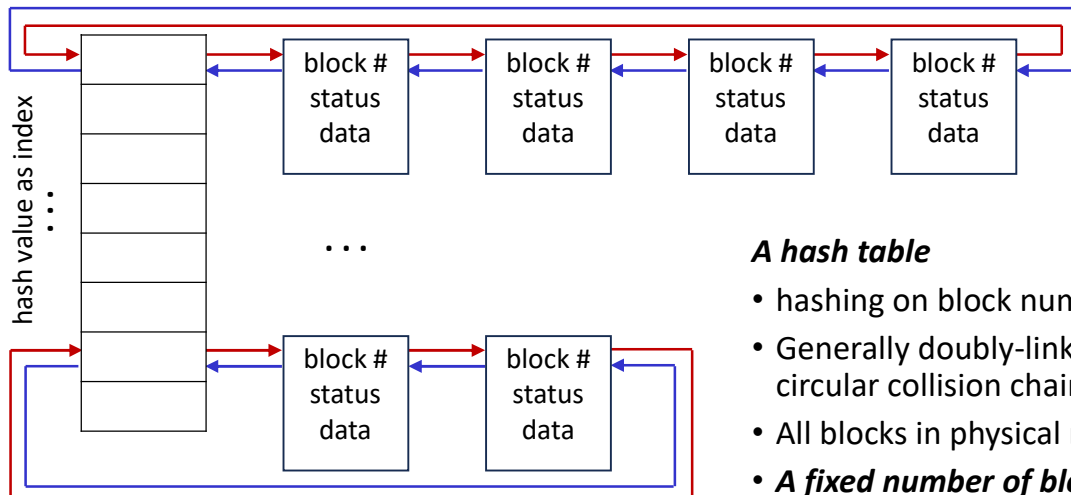
The File System Block Cache

A performance optimization to reduce disk reads and writes

- When a block is read from the disk
 - It is read it into some block buffer in the block cache
 - A copy of the block is kept (subject to capacity) in the cache for future use
- When the contents of a file system block is modified
 - Modify the copy of the block in the block cache
 - Just mark that block in the block cache as “dirty”
- When a block is evicted from the block cache (due to capacity)
 - If it is marked as “dirty”, it is written to disk before reusing the cache buffer

5

An Example Block Cache Structure



A hash table

- hashing on block number
- Generally doubly-linked circular collision chains
- All blocks in physical mem
- **A fixed number of blocks**

6

The sync() Kernel Call

```
void sync(void);
```

Causes all dirty file system data to be queued for writing to the disk

- Returns immediately after queuing all necessary disk writes
- Does not wait for completion of the writes
- Any user process can call sync() anytime it wants to

Classically a system process calls sync() periodically

- Example: “update” program on classical Unix calls sync() every 30 seconds

Helps some, but can't provide any persistence or consistency guarantees

The fsync() Kernel Call

```
int fsync(int fd);
```

Causes all dirty file system data for open file fd to be written to the disk

- Does not return until all writes have completed
- Includes all data for the file as well as “metadata” (e.g., inode) for it
- If the file system hard drive has internal caching (very common for hard disks)
 - Forces the data to be written through that caching too
 - Does not return until the data is really on the disk drive
- Useful to ensure the data and metadata for the file on disk is up-to-date
 - In case of a crash (e.g., power failure)

Ensuring File Contents and Metadata Is Up-to-Date

Consider the following simple program

- The `fsync(fd)` call ensures
 - All 10 blocks of data are on disk
 - The inode records, e.g., those 10 block numbers and records the file size of 40960
- But what about the name “data.txt”?
 - The directory is in data blocks separate from file data and separate from file inode
 - `fsync(fd)` does **not** ensure the name “data.txt” got recorded

```
int main(void)
{
    int fd;
    int i;
    char ch = 'X';

    fd = creat("data.txt", 0666);
    for (i = 0; i < 10 * 4096; i++)
        write(fd, &ch, 1);
    fsync(fd);
    exit(0);
}
```

Ensuring the Directory is Also Up-to-Date

Consider the following simple program

- To ensure name “data.txt” is also up-to-date in the directory
 - Use `fsync()` on fd open to the directory
- Can’t open for `O_RDWR` to get `fd2` since that would then let us directly modify the directory
- `fd2` just identifies the object we are doing `fsync()` on
 - The changes to add name “data.txt” have already taken place
 - `fsync(fd2)` forces those all of those changes to the directory to disk

```
int main(void)
{
    int fd;
    int i;
    char ch = 'X';

    fd = creat("data.txt", 0666);
    int fd2 = open(".", O_RDONLY);
    fsync(fd2);
    for (i = 0; i < 10 * 4096; i++)
        write(fd, &ch, 1);
    fsync(fd);
    exit(0);
}
```

Review: The mmap() Kernel Call

```
void *mmap(void addr[.length], size_t length, int prot, int flags,  
           int fd, off_t offset);
```

Map (part of) a file (or something else) into your virtual address space

- “fd” must be open to the file (or whatever) you want to map in from
- “addr” and “offset” must be multiple of PAGESIZE, but “length” need not be
- All pages covered by any part of [addr, addr+length) are mapped
 - if addr == NULL, the kernel chooses the virtual address
- “prot” should be PROT_NONE (no access) or the logical “or” of
 - PROT_READ – allow read (e.g., LOAD) accesses from the page
 - PROT_WRITE – allow write (e.g., STORE) accesses to the page
 - PROT_EXEC – allow CPU execution of instructions from the page
- mmap() returns the virtual address at which the memory was mapped

Review: The mmap() Kernel Call

The “flags” argument can specify many different things as the “or” of

- The sharing of the mapping with other processes, as either of
 - MAP_SHARED – Modifications to the memory **are** visible to other processes mapping this file and **are** eventually reflected in the file
 - MAP_PRIVATE – Modifications to the memory **are not** visible to other processes mapping to this file, and the file contents **is not** changed
- What to map to (the default is to map to a file)
 - MAP_ANONYMOUS (or MAP_ANON) – instead of a file, map to newly created “anonymous” memory (i.e., map to “nothing”)
- Restrictions on the virtual address to map to
 - MAP_FIXED – “addr” is normally treated as a “hint” by the kernel, but instead, the mapping must be exactly at virtual address addr

Review: The msync() Kernel Call

```
int msync(void addr[.length], size_t length, int flags);
```

Synchronize the file with the memory mapping

- Make sure the disk copy is up-to-date with changes in memory copy
- “addr” must be a multiple of PAGESIZE, but “length” need not be
- All pages covered in any part by [addr, addr+length) are updated
- “flags” should be either
 - MS_ASYNC – ***asynchronous*** update, returns immediately
 - MS_SYNC – ***synchronous*** update, ***does no return until updated on disk***
- Happens implicitly when munmap() is used on these pages
 - And thus happens implicitly on _exit()

Behavior of mmap() and MAP_SHARED

- All processes mapping the same area of some file MAP_SHARED immediately see each other’s changes
 - They are actually mapped to the same ***physical*** pages, so there is no way they couldn’t immediately see each other’s changes
- msync() has some similarity to sync() and to fsync()
 - With MS_ASYNC, it is more like sync() in just queueing the writes to disk
 - With MS_SYNC, it is more like fsync() in waiting for writes to complete
 - Both MS_ASYNC and MS_SYNC just apply to the given addr and length range of bytes of the file
 - Neither applies to the file’s metadata (e.g., inode updates)

mmap() MAP_SHARED is Widely Misunderstood

Example – ChatGPT got it wrong!

When asked

- Can you write some simple C programs that allows you to see that data is not written to a file if you don't use msync after writing to an mmaped file?

ChatGPT suggested the following two (simplified) programs to show this

- And provided an explanation of the two programs

Programs Provided by ChatGPT

```
int fd;
char *mapped;

fd = creat("test_file.txt", 0666);
mapped = mmap(NULL, FILE_SIZE,
              PROT_READ|PROT_WRITE,
              MAP_SHARED, fd, 0);
strcpy(mapped, "Hello world!\n");
printf("Data written: %s\n", mapped);

// no msync() or any other flushing:
sleep(10); // Run Program 2

munmap(mapped, FILE_SIZE);
close(fd);
```

```
int fd;
char *mapped;

fd = open("test_file.txt", O_RDONLY);
mapped = mmap(NULL, FILE_SIZE,
              PROT_READ, MAP_SHARED, fd, 0);

// will likely print nothing or garbage:
printf("Data read: %s\n", mapped);

munmap(mapped, FILE_SIZE);
close(fd);
```

ChatGPT's (Incorrect!) Explanation

Step 1

Run Program 1 to write data to the memory-mapped file (test_file.txt). Since `msync()` isn't called, the data is written to the memory-mapped region but not flushed to the file on disk.

Step 2

Run Program 2, which attempts to read from the file. Since Program 1 didn't flush the data with `msync()`, Program 2 **will not see the changes that Program 1 made**. The output will likely show empty or outdated data from the file (e.g., empty string or garbage data).

This is incorrect – Program 2 will always see all of Program 1's changes!

Doing `read()` and `write()` vs. `mmap()` Changes

- Doing `read()` and/or `write()` uses the file system block cache
 - Data moves between the file and the block cache
 - Other processes also doing `read()` and/or `write()` on the same file see each other's changes since they are sharing entries in the block cache
- Doing `mmap()` uses physical pages outside of the block cache
 - These physical pages are mapped into the process's virtual address space
 - File blocks only brought into memory when accessed (demand paging)
- Processes doing `read()/write()` vs. processes doing `mmap()` don't see each other's changes! (Except as they get evicted/reloaded due to lack of space)
- And can easily end up with two copies of the same data in physical memory!

Unified Block Cache and Virtual Memory

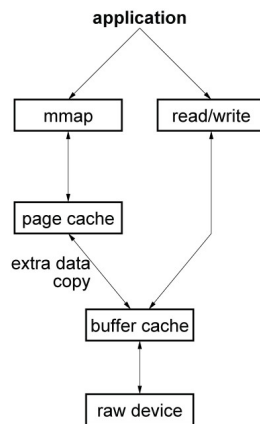
Fully integrate the block cache management and virtual memory

- Each page of physical memory holding a page of MAP_SHARED data (or, e.g., a shared text page) is effectively the same as a block in the block cache
- Block cache and virtual memory share essentially all of physical memory
- Allows better dynamic allocation of how physical memory pages are used
- Now only one copy of any file block anywhere in physical memory
- And processes doing read()/write() and/or mmap() accesses immediately see each other's changes

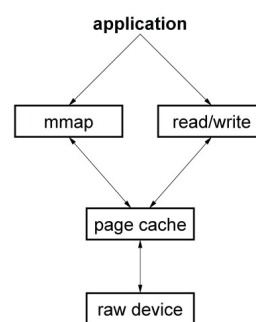
19

Unified Block Cache and Virtual Memory

Without Unified Cache



With Unified Cache



UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD, Chuck Silvers, USENIX 2000

20

File System Access over a Network

Adds additional layers of buffering and latency

- Example: CLEAR home directory access over NFS (Network File System)
- Local kernel writes data over the network rather than to local disk
- Remote server machine receives data from network and writes to local disk
- Both machines have buffering at the network level
- Data or metadata changes may take longer to end up on disk

- `fsync()` is defined to write all the way to the remote disk before returning
- Also when the last local file descriptor to it is closed
- Otherwise, data and metadata changes should eventually get to the disk, but it can be a “long” time