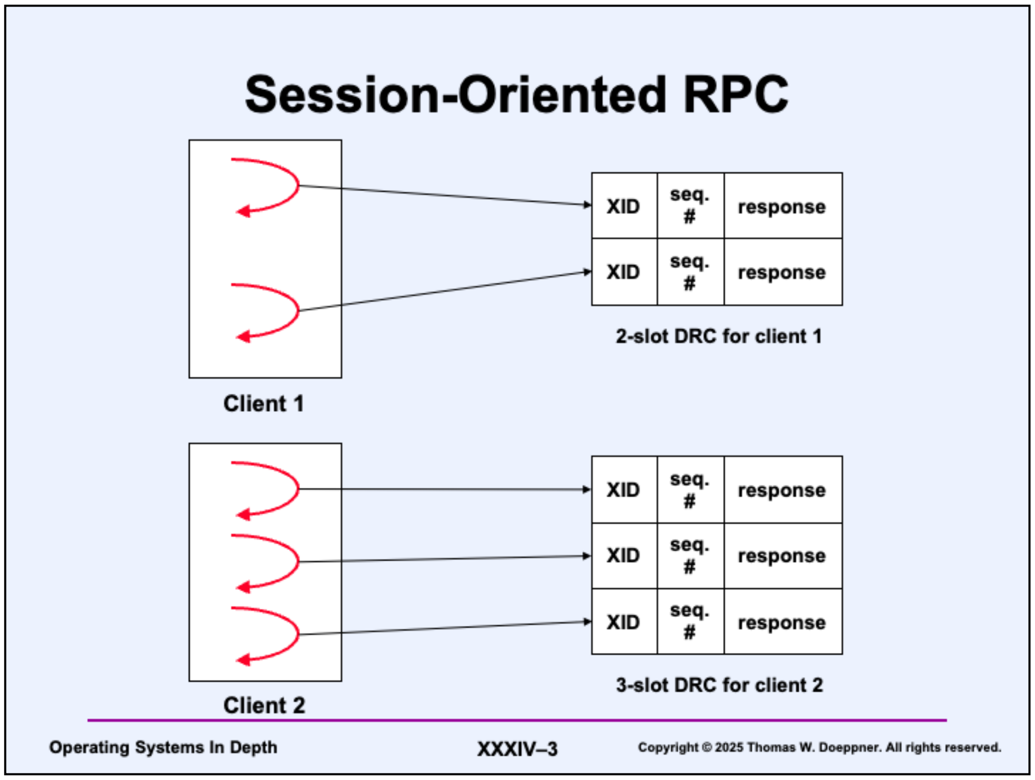
Remote Procedure Call Protocols (2)

What’s Wrong?

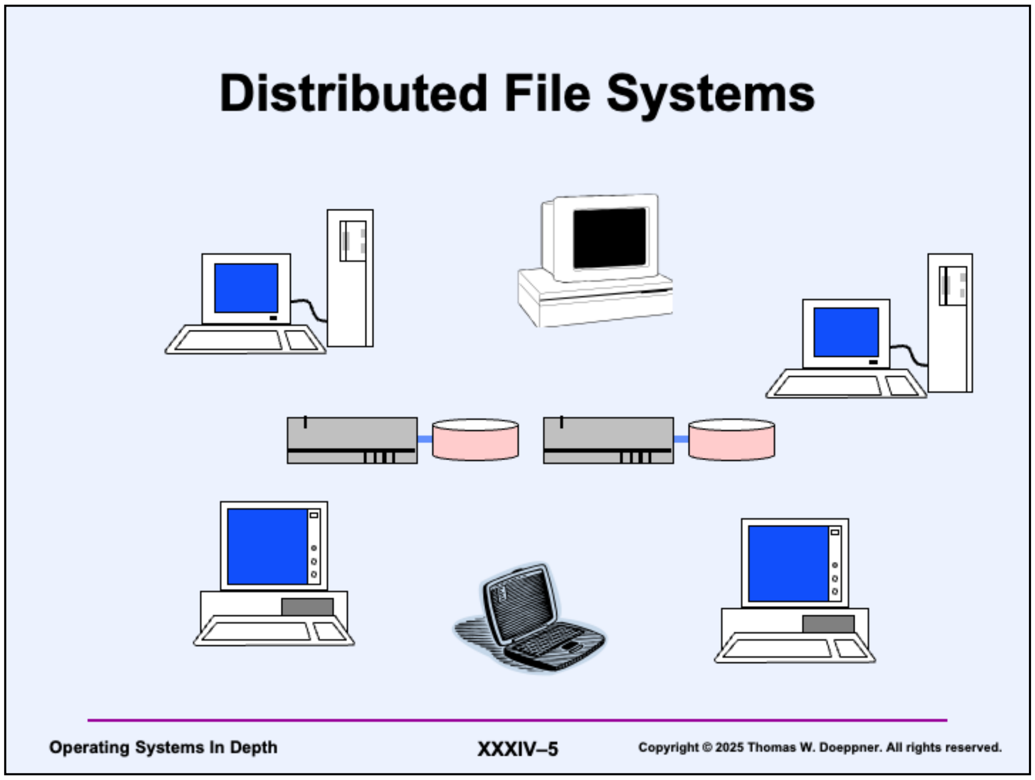
- The problem is the duplicate request cache (DRC)
  - it’s necessary
  - but when may cached entries be removed?



Session-Oriented RPC

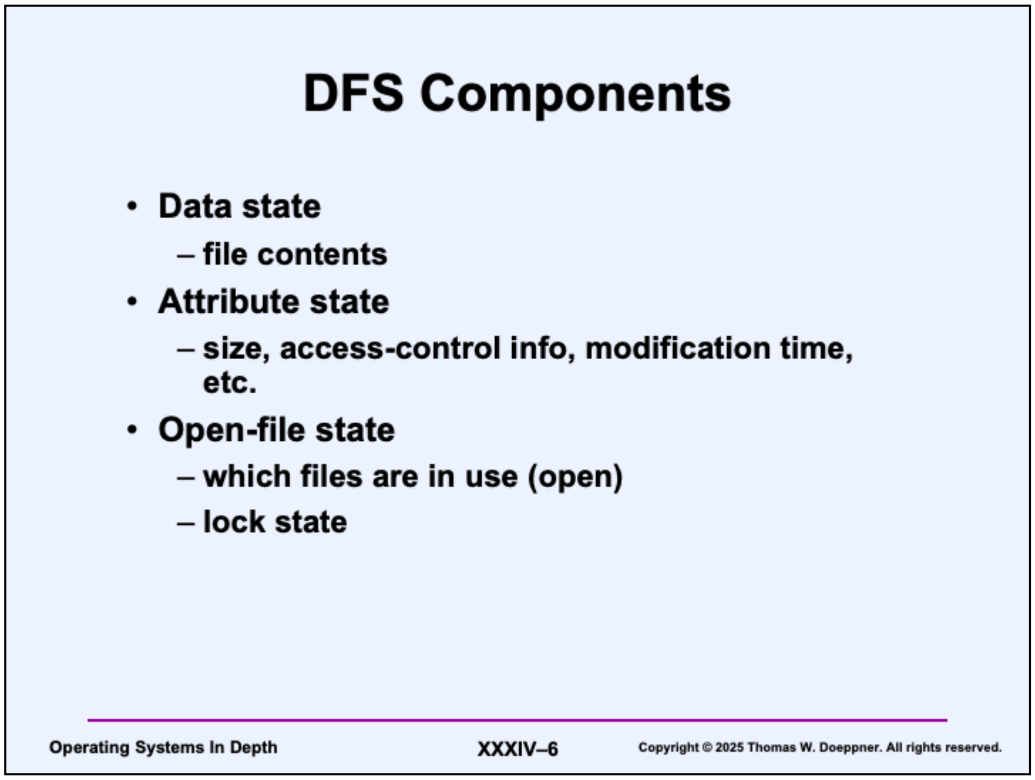
In session-oriented RPC, the client, as part of setting up a **session** with the server, lets the server know ahead of time the maximum number of concurrent requests it will issue; the server then creates that number of **channels** for it. Each client request then provides a channel number and a sequence number. Only one request at a time may be active on a channel. Requests on each channel carry consecutive sequence numbers. The server maintains a separate DRC entry for each channel of each client, containing the XID, sequence number, and response, as shown in the slide. An entry's contents are not deleted until a request arrives with the next sequence number for that channel.

Distributed File Systems Part 1



Distributed File Systems

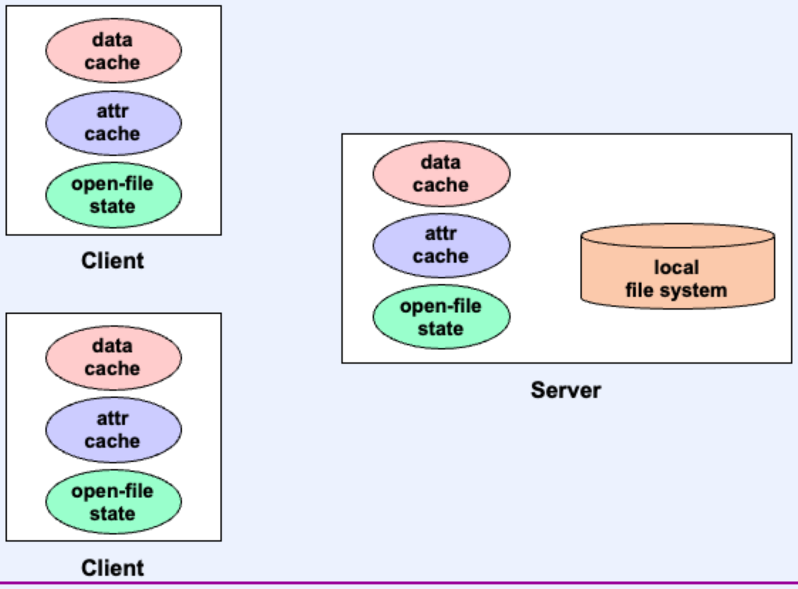
File systems are certainly important parts of general-purpose computer systems. They are responsible for the storage of data (organized as files) and for providing a means for applications to store, access, and modify data. Local file systems handle these chores on individual computers; distributed file systems handle these chores on collections of computers. In the typical design, distributed file systems provide a means for getting at the facilities of local file systems. What is usually desired of a distributed file system is that it be access transparent: programs access files on remote computers as if the files were stored locally. This rules out approaches based on explicit file transfer, such as the Internet's FTP (file transfer protocol) and Unix's rcp and scp (remote copy).



DFS Components

- Data state
  - file contents
- Attribute state
  - size, access-control info, modification time, etc.
- Open-file state
  - which files are in use (open)
  - lock state

Possible Locations



Quiz 1

We'd like to design a file server that serves multiple Unix client computers. Assuming no computer ever crashes and the network is always up and working flawlessly, we'd like file-oriented system calls to behave as if all parties were on a single computer.

a) It can't be done

b) It can be done, but requires disabling all client-side caching

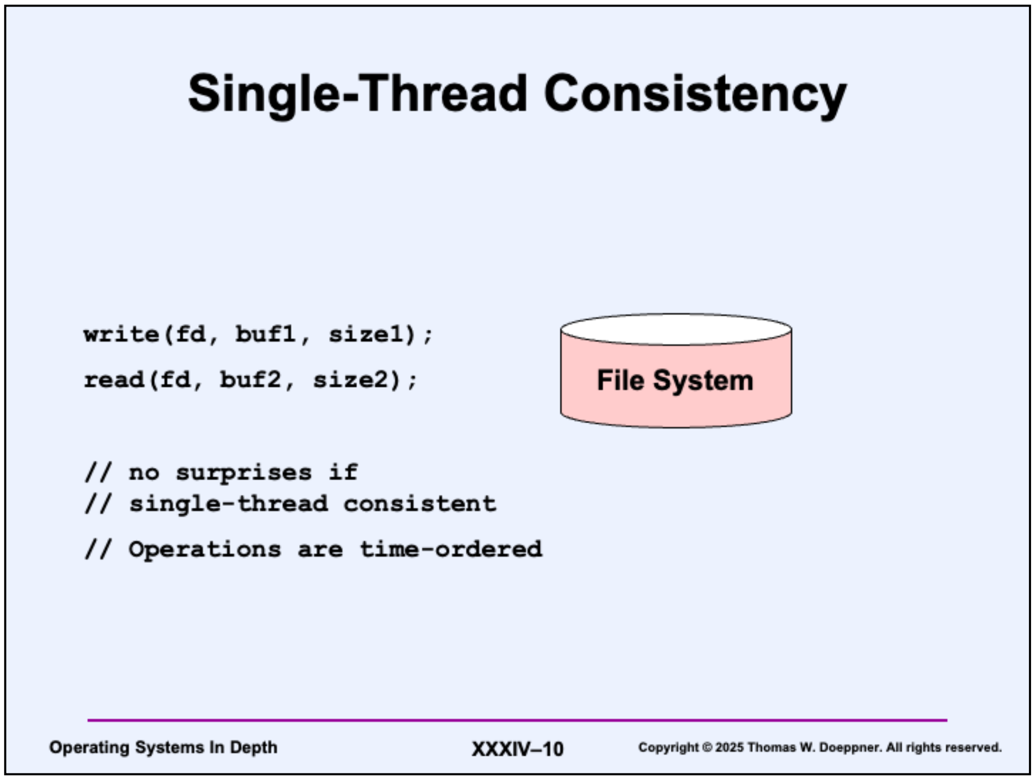
c) It can be done, but sometimes requires disabling client-side caching

d) It can be done, irrespective of client-side caching

Guiding Principle

Principle of least astonishment (PLA)

- people don't like surprises, particularly when they come from file systems



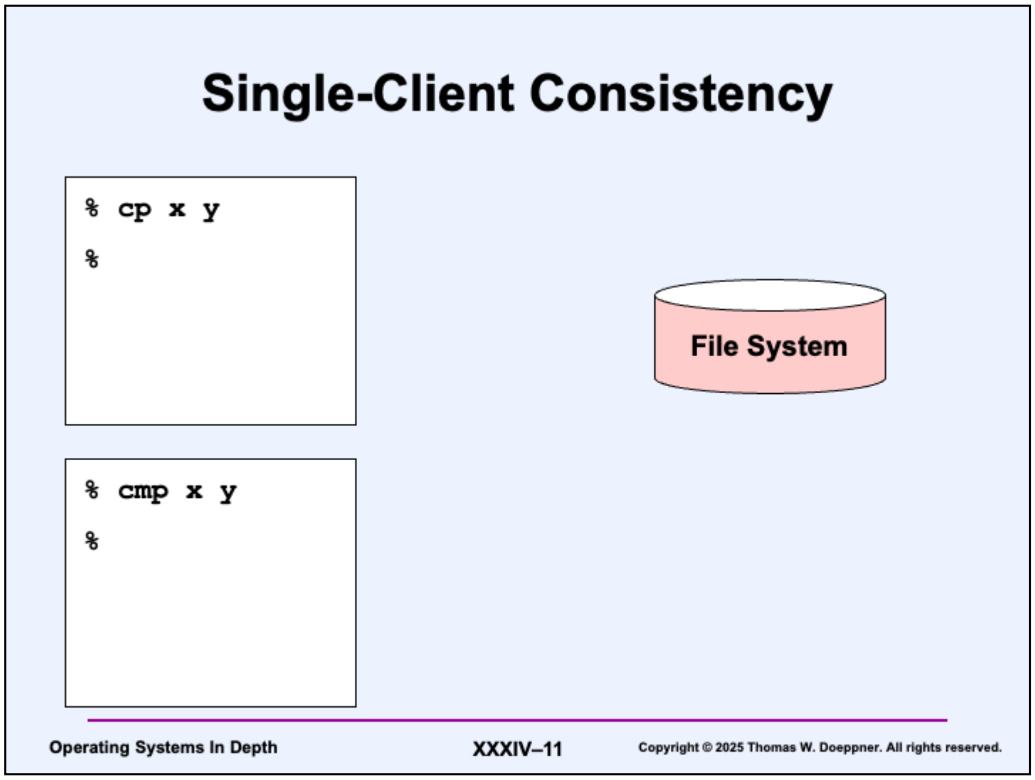
Single-Thread Consistency

```
write(fd, buf1, size1);
read(fd, buf2, size2);

// no surprises if
// single-thread consistent

// Operations are time-ordered
```

File System

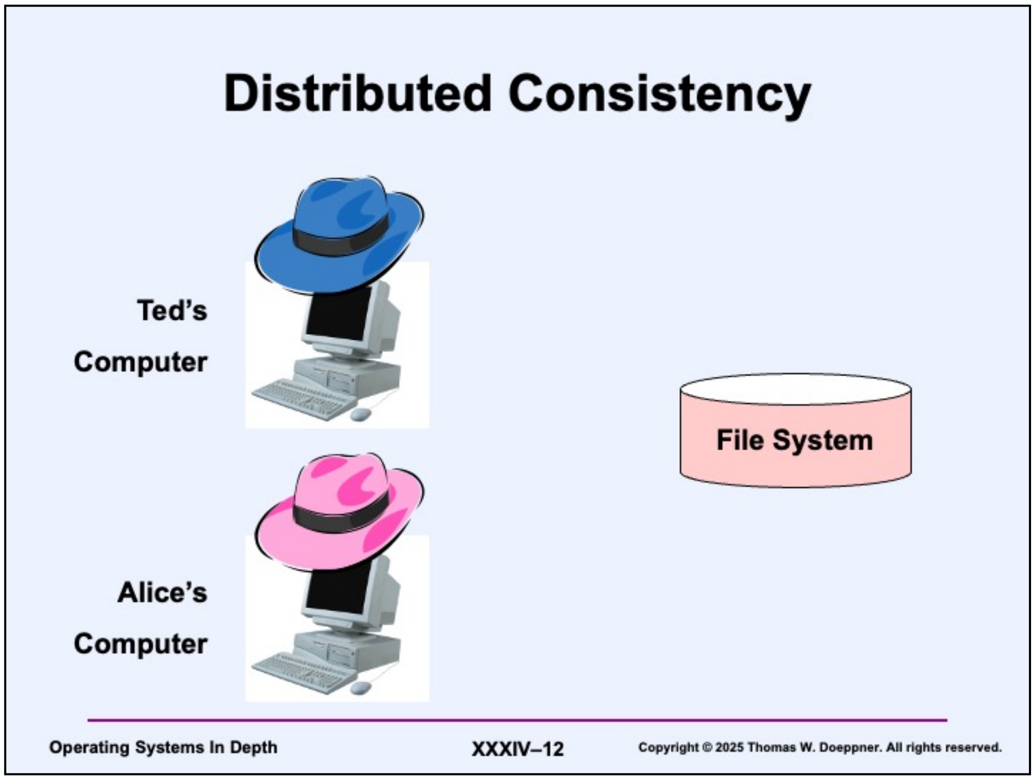


Single-Client Consistency

```
% cp x y
%
```

```
% cmp x y
%
```

Here the operations are done on separate processes on the same computer. The "cp" takes place just before the "cmp".



Distributed Consistency

Ted's Computer
Alice's Computer
File System

Ted and Alice are using different computers that both access a file system on a server.

Strict Consistency

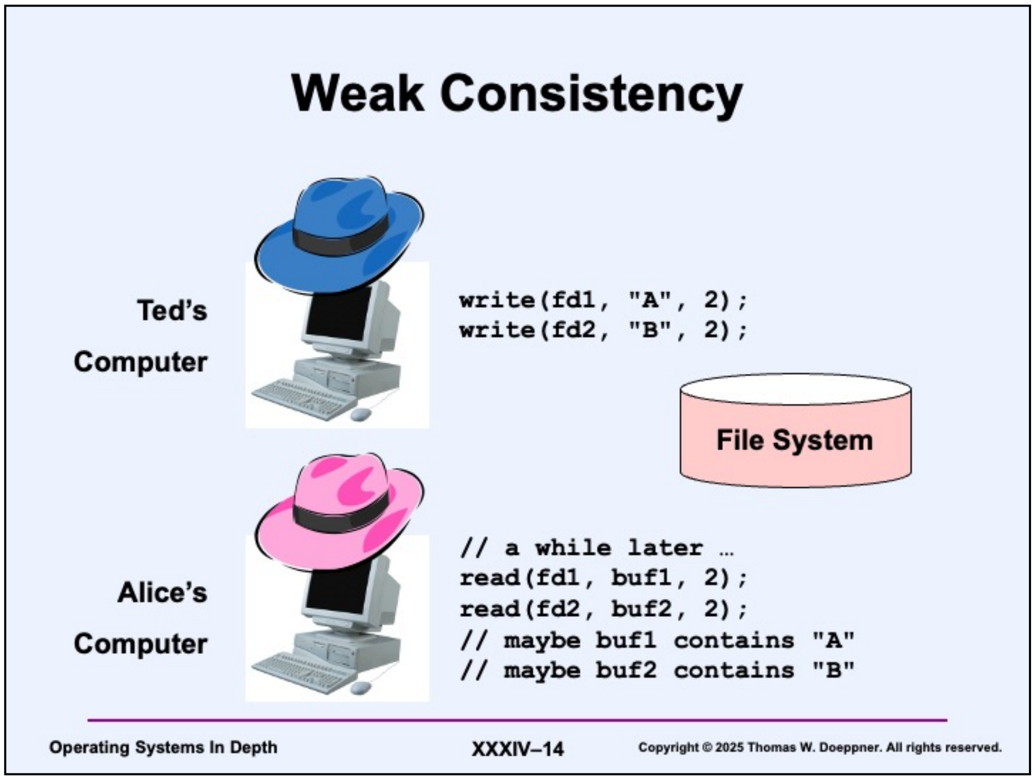
Ted's Computer

```
write(fd1, "A", 2);
write(fd2, "B", 2);
```

File System

Alice's Computer

```
// an instant later …
read(fd1, buf1, 2);
read(fd2, buf2, 2);
// buf1 contains "A"
// buf2 contains "B"
```



Weak Consistency

Ted's Computer
Alice's Computer
File System

```
write(fd1, "A", 2);
write(fd2, "B", 2);
```

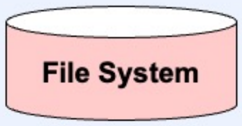
```
// a while later …
read(fd1, buf1, 2);
read(fd2, buf2, 2);
// maybe buf1 contains "A"
// maybe buf2 contains "B"
```

Changes made to a file by one client will eventually be seen by others, though not necessarily in the order they were made.

Sequential Consistency

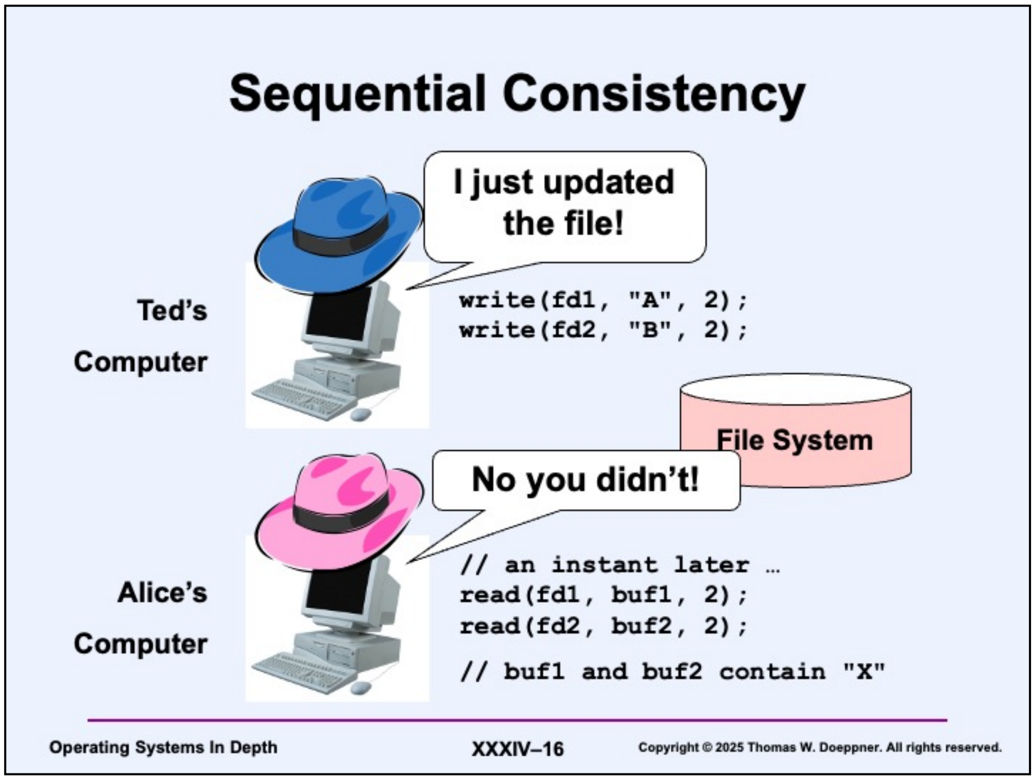
Ted's Computer

```
write(fd1, "A", 2);
write(fd2, "B", 2);
```



Alice's Computer

```
// an instant later …
read(fd1, buf1, 2);
read(fd2, buf2, 2);
// if buf2 contains "B"
// then buf1 contains "A"
```



Sequential Consistency

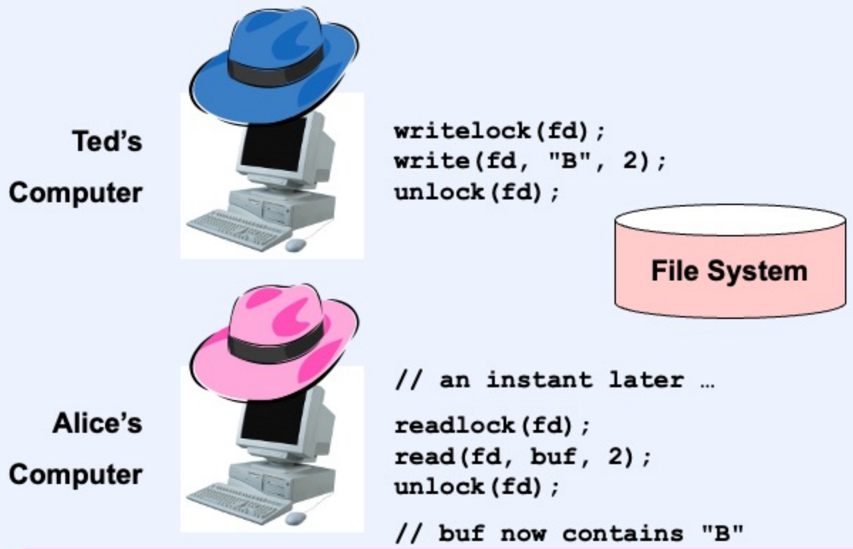
I just updated the file!
Ted's Computer
File System
No you didn't!
Alice's Computer

```
write(fd1, "A", 2);
write(fd2, "B", 2);
```

```
// an instant later …
read(fd1, buf1, 2);
read(fd2, buf2, 2);
// buf1 and buf2 contain "X"
```

Assume that both files initially contain “X”.

Entry Consistency

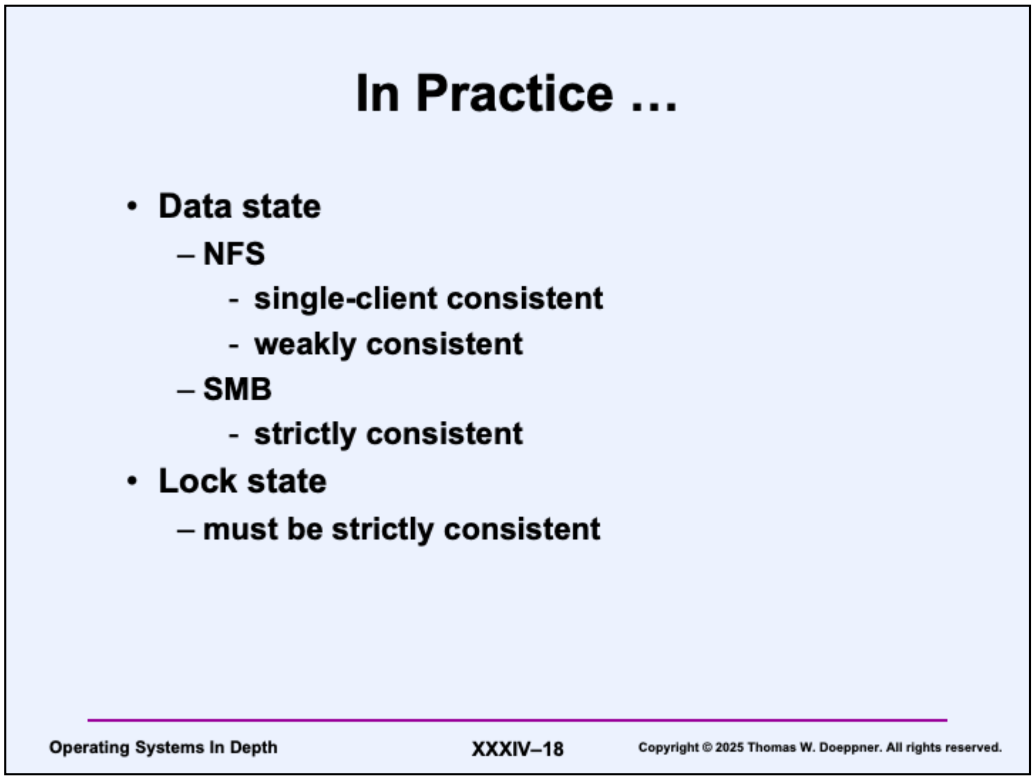


```
writelock(fd);
write(fd, "B", 2);
unlock(fd);
```

```
// an instant later …

readlock(fd);
read(fd, buf, 2);
unlock(fd);

// buf now contains "B"
```



In Practice …

- Data state
  - NFS
    - single-client consistent
    - weakly consistent
  - SMB
    - strictly consistent
- Lock state
  - must be strictly consistent

Thursday morning, November 17th
At 7:00 a.m.

Maytag, the department's central file server, will be taken down to kick off a filesystem consistency check.

Linux machines will hang.

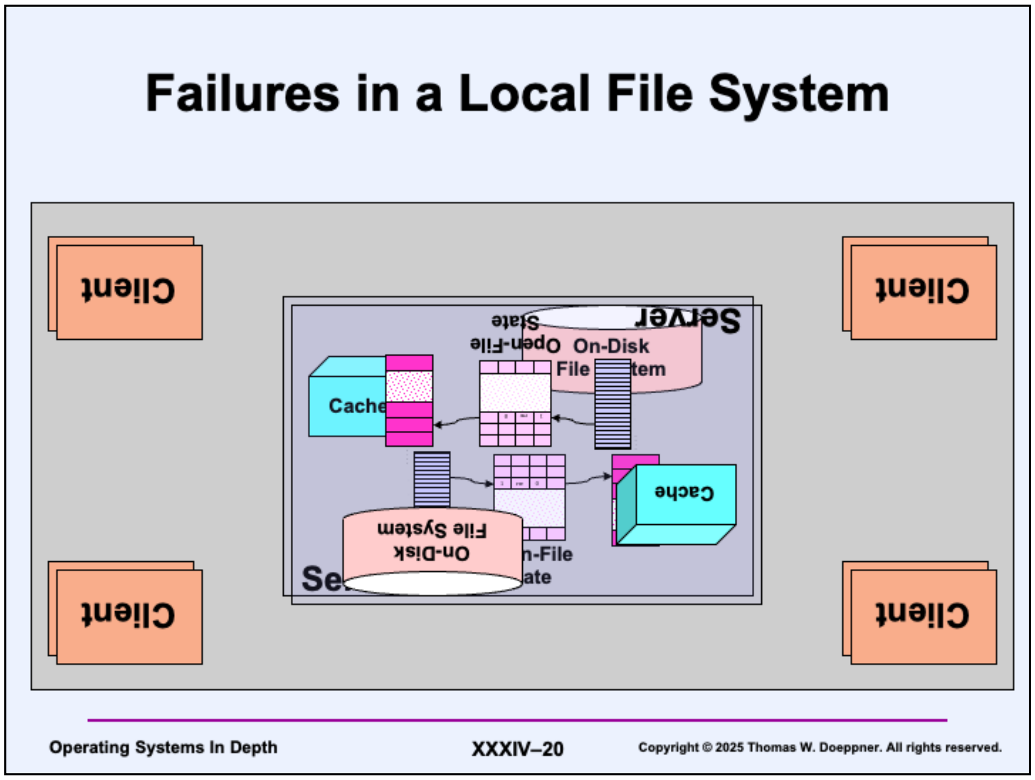
All Windows users should log off.

Normal operation will resume by 8:30 a.m. if all goes well.

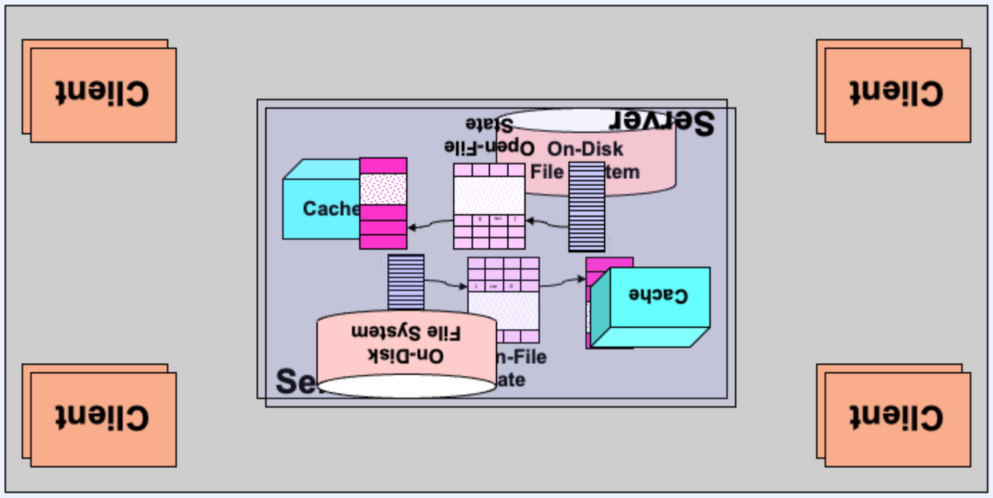
All windows users should log off before this time.

Questions/concerns to problem@cs.brown.edu

(Note that the November 17 in question was in 2005.)

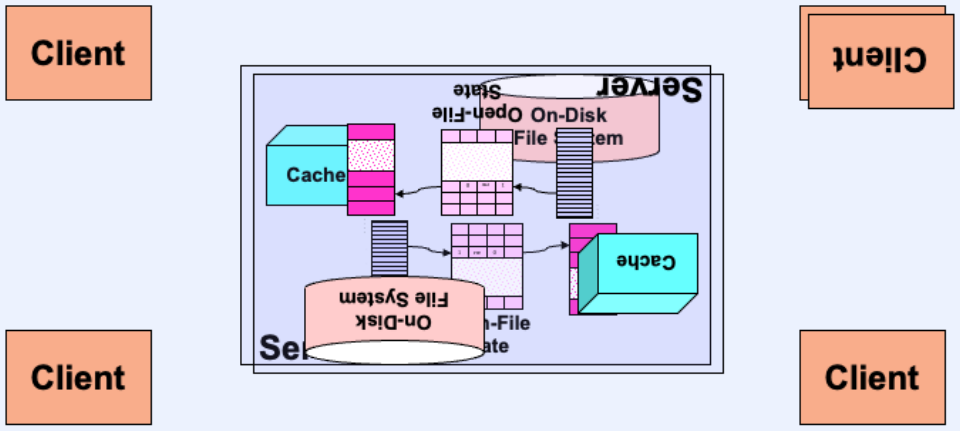


Failures in a Local File System



What we're accustomed to with local file systems is that, in the event of a crash, everything goes down. This is simple to deal with.

Distributed Failure



In a distributed system, if the server crashes, there is no inherent reason for clients to crash as well. Assuming there was no damage done to the on-disk file system, client processes might experience a delay while the server is down, but should be able to continue execution once the server comes back up, as if nothing had happened. The crash of a client computer is bad news for the processes running on that computer, but should have no adverse effect on the server or on other client computers. We'd like the effect to be as if the client processes on the crashed computer had suddenly closed all their files and terminated.

Quiz 2

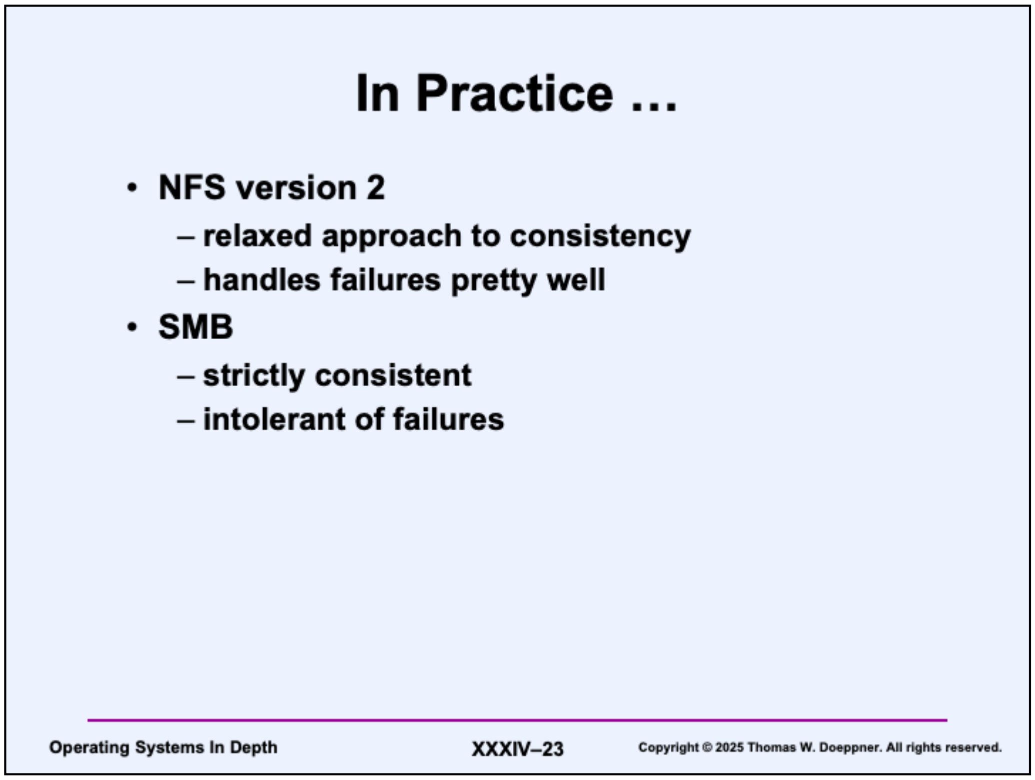
We'd like to design a file server that serves multiple Unix client computers, but we now realize we must cope with failures. Which one of the following is not true?

a) At least one of the following statements is false

b) If we relax Unix system-call semantics a bit, this is easy

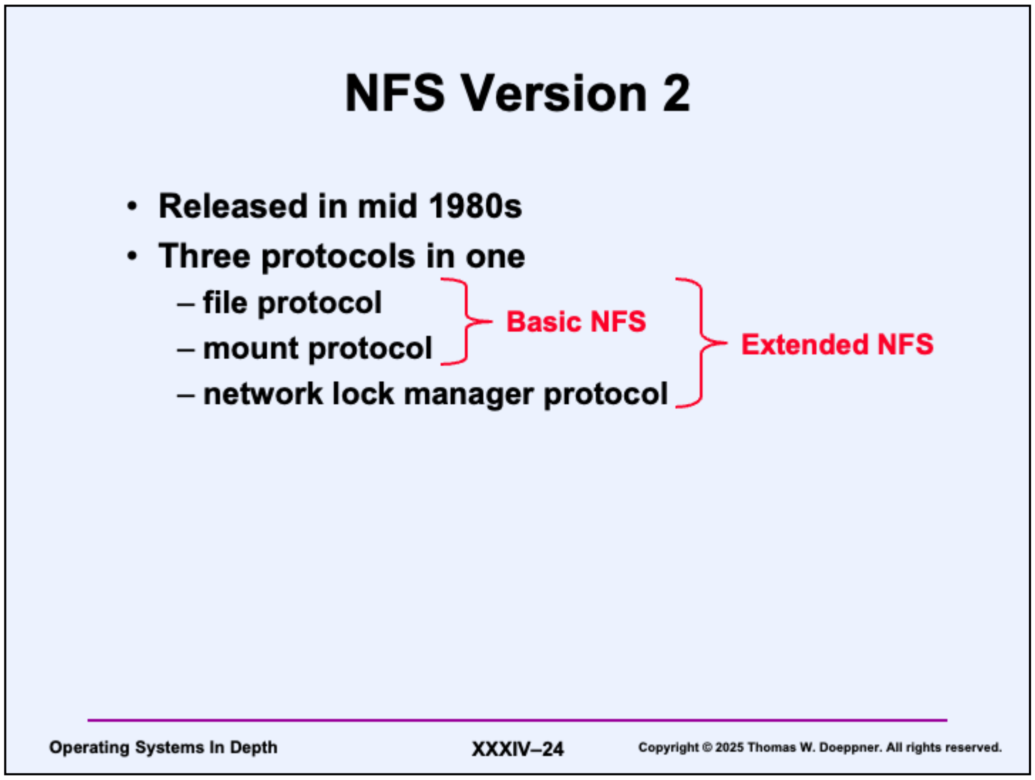
c) If we don't relax Unix system-call semantics, it's doable, but we need to introduce some new error messages for certain situations

d) There are failure modes that can't possibly occur if all parties are on the same computer



In Practice …

- NFS version 2
  - relaxed approach to consistency
  - handles failures pretty well
- SMB
  - strictly consistent
  - intolerant of failures



NFS Version 2

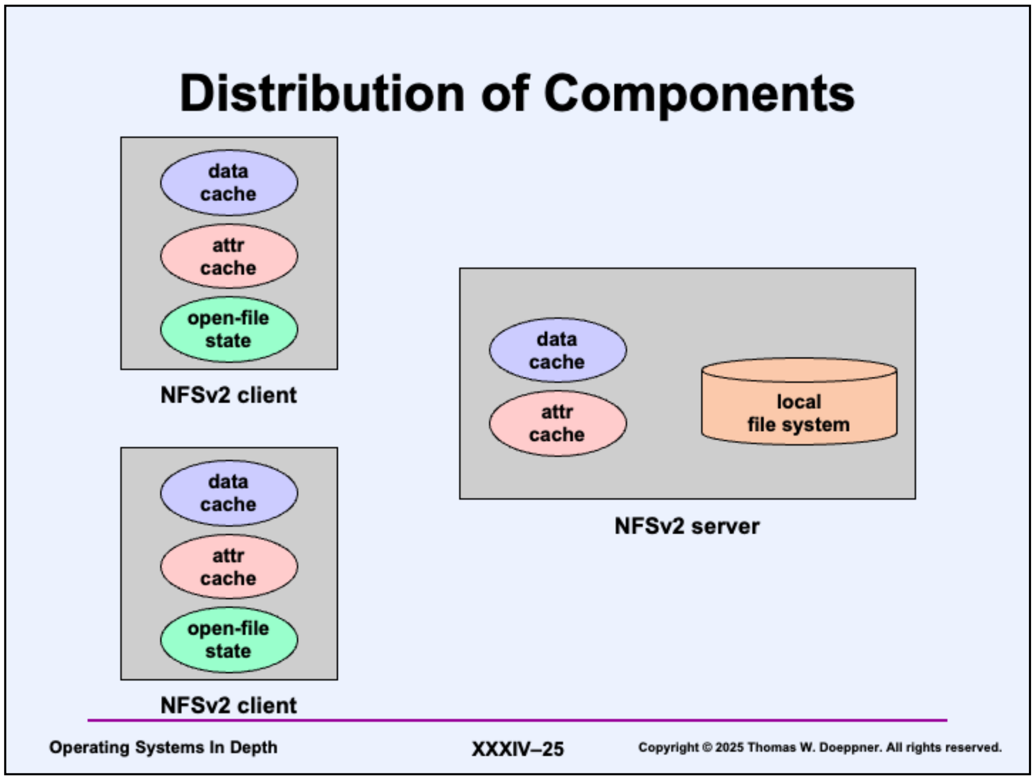
- Released in mid 1980s
- Three protocols in one
  - file protocol
  - mount protocol
  - network lock manager protocol

Basic NFS
Extended NFS

NFS consists of three component protocols: a **mount protocol** for making collections of files stored on servers available to client nodes, the **NFS file protocol** for accessing and manipulating files and directories, and a **network lock manager** (NLM) for locking files over the network and recovering lock state after failures.

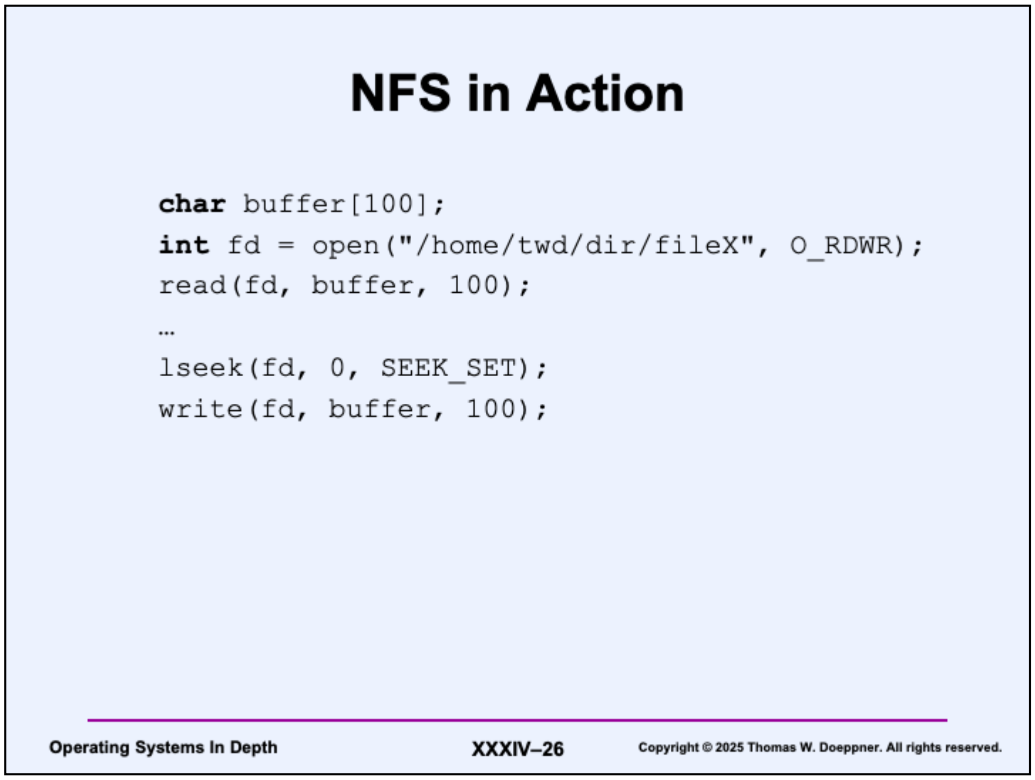
An NFS server gives its clients access to one or more of its local file systems, providing them with opaque identifiers called **file handles** to refer to files and directories. Clients use a separate protocol, the **mount protocol**, to get a file handle for the root of a server file system, then use the **NFS protocol** to follow paths within the file system and to access its files, placing simple remote procedure calls to read them and write them. A third protocol, the **network lock manager protocol** (NLM), was added later and may be used, if desired, to synchronize access to files. All communication between client and server is with ONC RPC.

Note that we run NFS versions 3 and 4 at Brown CS. Version 3 is very similar to version 2.



Distribution of Components

Note that the NFSv2 server does not maintain open-file state–for that reason it's said to be **stateless**.



NFS in Action

```
char buffer[100];
int fd = open("/home/twd/dir/fileX", O_RDWR);
read(fd, buffer, 100);
…
lseek(fd, 0, SEEK_SET);
write(fd, buffer, 100);
```

The thread opens a file, reads the first 100 bytes, then writes 100 bytes at the beginning of the file, replacing the previous data there. The kernel data structures representing the open file are just as they would be if the file were local. The only difference from the client operating system's point of view is that the actual operations on the file are handled by the NFSv2 client code rather than by a local file system such as ext4.

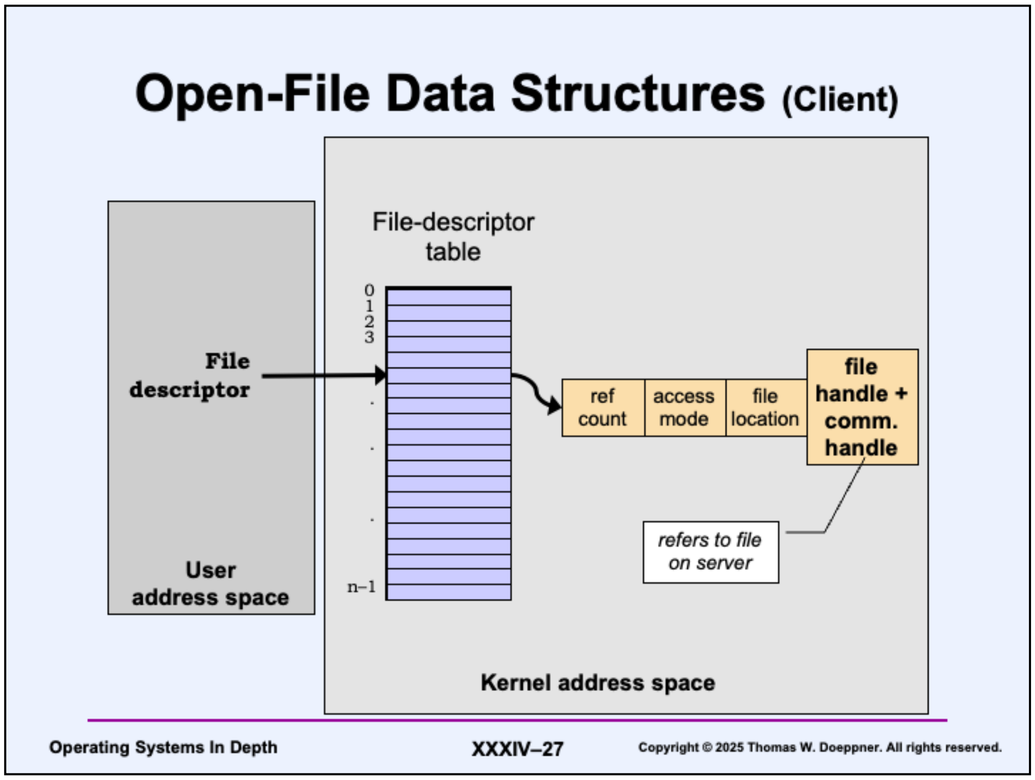
The NFSv2 server has its own directory hierarchy. It exports a number of subtrees of this hierarchy to its clients. The client's operating system has mounted one of them on the local directory /home/twd. It did this by first placing a remote procedure call, using the mount protocol, to the server, obtaining a file handle for the root of this subtree. Any attempt to access the directory /home/twd will be interpreted as an attempt to access the remote directory represented by the file handle.

Thus, to follow the path /home/twd/dir/fileX, our thread, executing in the client operating system, follows the path as far as /home/twd and discovers that a remote file system is mounted there. It places a remote procedure call to the NFSv2 server's **lookup** routine, passing it the file handle for the root as well as the pathname /dir/fileX. The server returns a file handle for fileX.

Just as it would for local file, the client operating system records the fact that fileX is open. However, rather than having an inode or equivalent to represent the file, it uses the file handle together with some sort of communication handle representing the remote server.

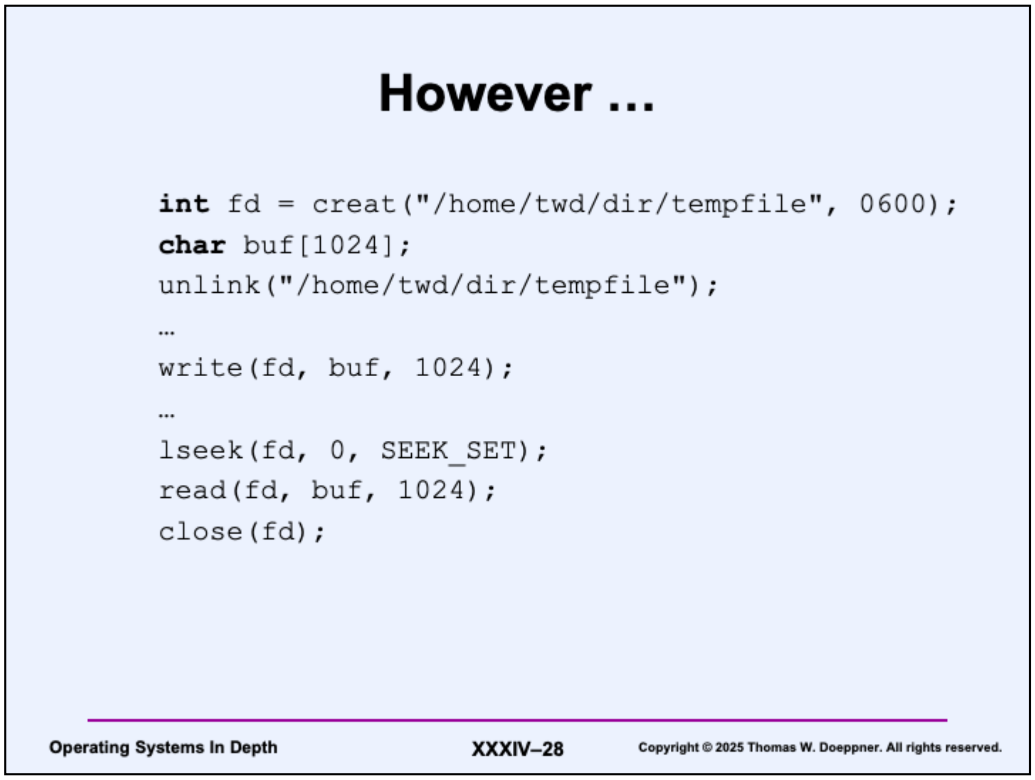
When the thread performs the **read**, it, within the client operating system, places a remote procedure call passing the file handle, the offset within the file (0), and the length (100) to the server's read routine. The server then returns the 100 bytes from that location.

The **lseek** call sets the local file offset to 0, but causes no request to be sent to the server. When the thread calls **write**, it places a remote procedure call containing the file handle, the current offset (now 0), the data, and the length to the server's write routine. The server returns an indication of successful completion of the operation.



Open-File Data Structures (Client)

Note that the fact that a file is open is known only on the client. The server is stateless (with respect to client access to file systems) and doesn't keep track of who has what open and how.



However …

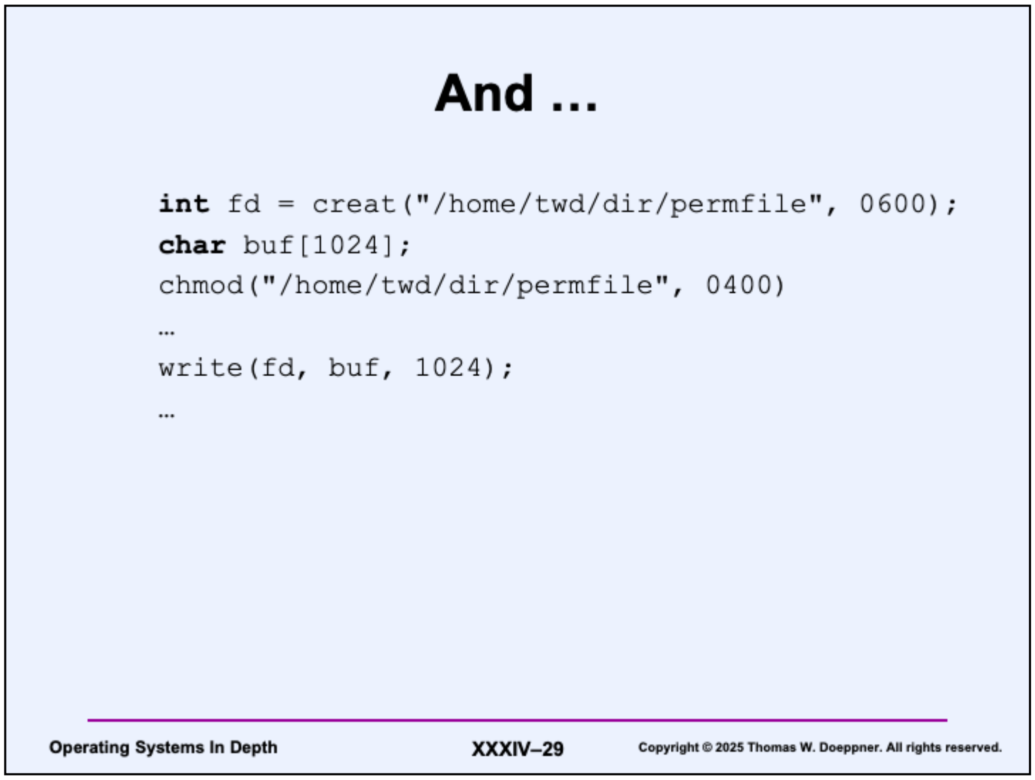
```
int fd = creat("/home/twd/dir/tempfile", 0600);
char buf[1024];
unlink("/home/twd/dir/tempfile");
…
write(fd, buf, 1024);
…
lseek(fd, 0, SEEK_SET);
read(fd, buf, 1024);
close(fd);
```

The file /home/twd/dir/tempfile is created, then its directory entry is immediately removed. If this were done on a local file system, the file itself would continue to exist, even though there no longer is a directory entry referring to it—that the file is open causes it to continue to exist. This code sequence is rather common in Unix programs, it is a standard technique for creating a temporary file—one that is guaranteed to disappear once the process terminates: as soon as the file is closed, there is no longer a reference to it and the operating system deletes it.

Since NFSv2 servers are stateless, they do not keep track of whether files are open. Thus the **unlink** request in the code above would remove the last link to the file and leave it with no other references—the file would then be deleted, much to the surprise of the client.

NFSv2 relies on the client to keep track of such state information and to modify its requests to the server accordingly. In our example, the client would realize that sending the server an unlink request for an open file isn't in its best interests. So, rather than sending the server an unlink request on an open file, it sends it a rename request, changing the file's name to a special name that does not show up in normal directory searches (in Unix, this is accomplished by having the name start with "."). Thus the file no longer appears to exist by its original name, yet the client's file handle still refers to it. When the client process closes the file, then an unlink request is sent to the server, removing the file's special name from the directory and thus finally deleting the file.

This solution clearly is not perfect: it works only if the file is open on the same machine as the one doing the unlink. In practice this is the only case that matters, so it works well enough. But what happens if the client, after renaming a file, crashes before the file is closed? The result is that the renamed file is not deleted. It remains on the server. The server must periodically check for such orphaned files and delete them.



And ...

```
int fd = creat("/home/twd/dir/permfile", 0600);
char buf[1024];
chmod("/home/twd/dir/permfile", 0400)
…
write(fd, buf, 1024);
…
```

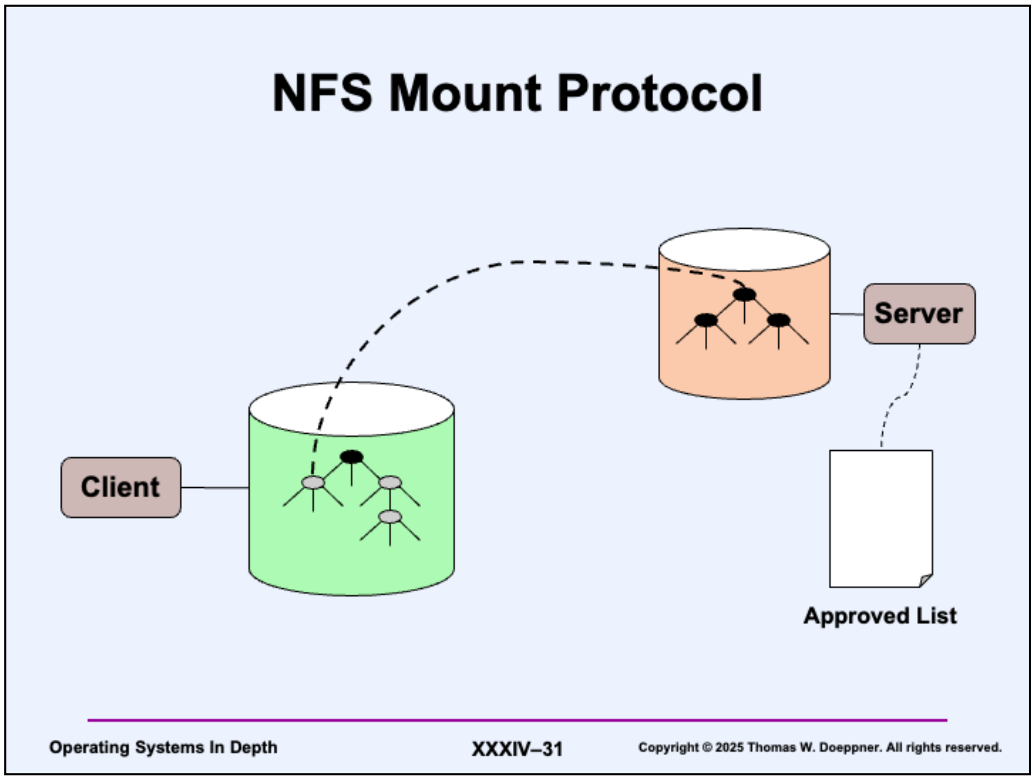
Here the client has created a file, giving the owner read-write permission. It then changes the permissions to read-only, since it doesn't intend for the file's contents to change after initialization. It then initializes the file by writing to it.

On a local file system, since the file was opened read-write and the process has the appropriate file handle, the write will succeed. But over NFS, since the server has no state information describing how the process has opened the file, the server checks permissions on each access. Since the file is now read-only, the write will fail.

Since this is an example of something the programs actually do (and depend on), NFS must somehow make this example work. So, NFS servers allow the owner of a file to read from it and write to it, regardless of the current permissions. The client OS, when the open is performed, must check if the process is allowed the desired access (even if its user is the owner), and allow the open to succeed only if the access is permitted.

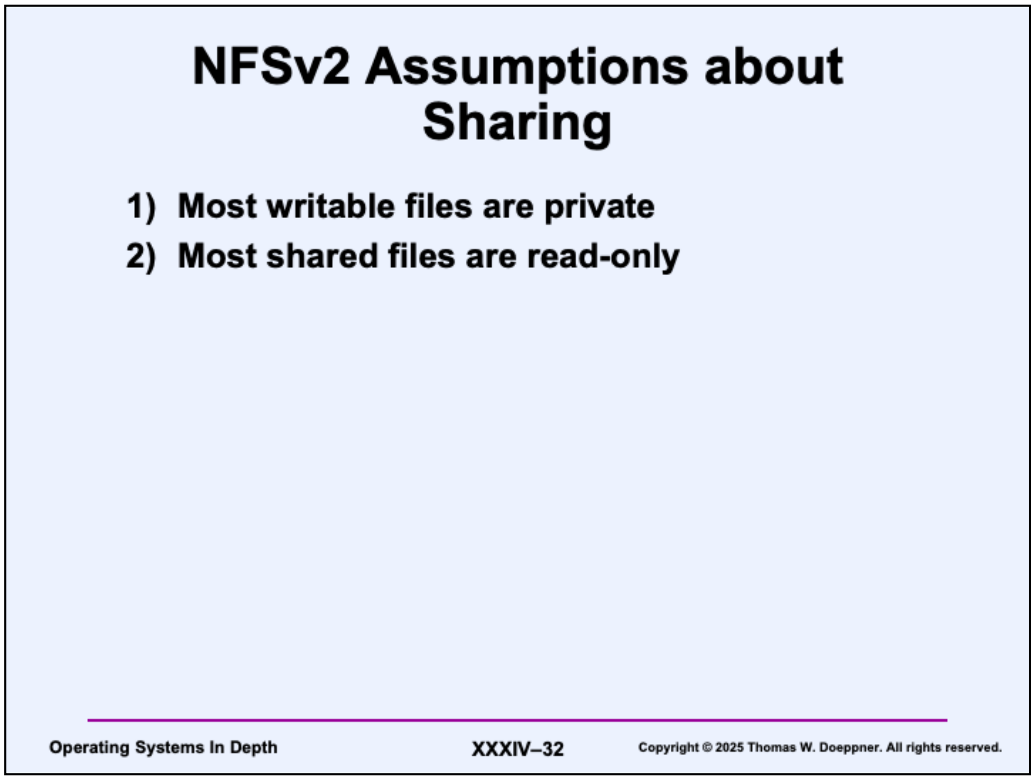
RPC Semantics

- All requests done with ONC RPC
- Most are idempotent
- A few aren't
  - e.g. unlink
- Made *reasonably* reliable with DRC
  - susceptible to Byzantine routers and poorly timed crashes
    - crashes affect ability to handle retransmitted requests correctly



NFS Mount Protocol

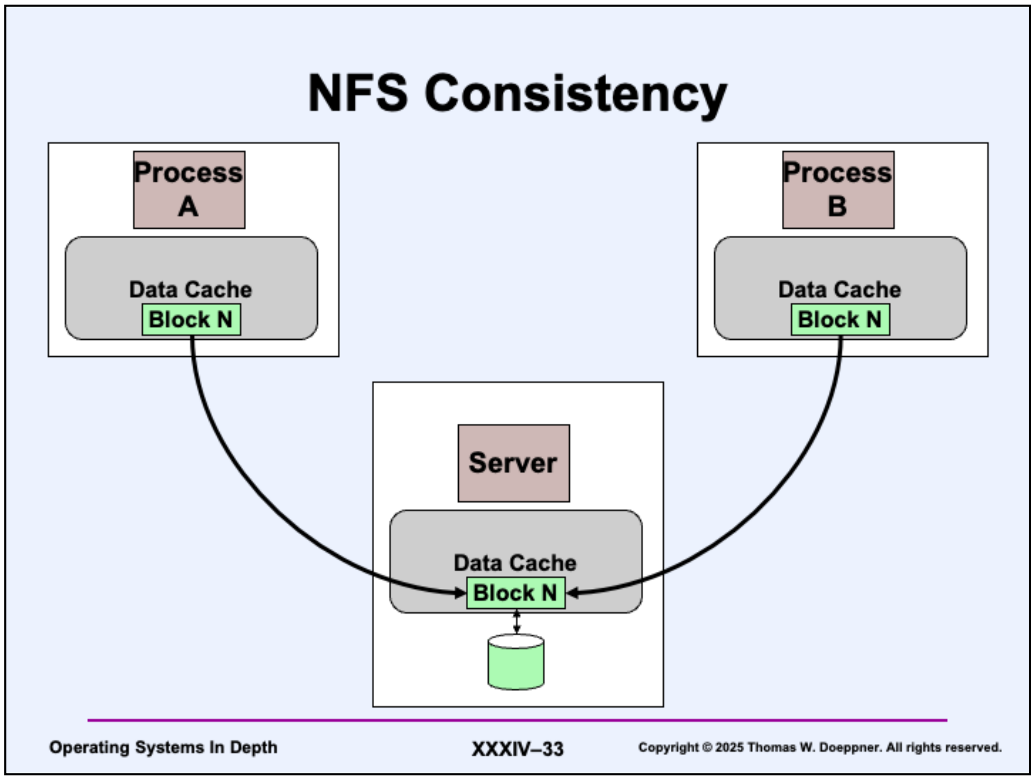
Servers divide their files up into disjoint collections called **file systems**, each of which contains a rooted directory tree naming all of its members. Servers, as specified in local configuration files, specify which file systems, or subtrees within a file system, are available to which clients. A client can then **mount** a remote file system. This entails superimposing the root of the remote file system on top of a directory in the client's current naming tree. The root of the remote file system (the mounted file system) effectively replaces the mounted-on directory. Thus the remote file system is attached to the client's naming tree at the mounted-on directory (and the previous contents of that directory are invisible as long as the remote file system remains mounted). The mount protocol provides some security (by restricting which clients are allowed to mount a file system) and gives the client node the means for fitting remote file systems into its file-system name space.



NFSv2 Assumptions about Sharing

1) Most writable files are private
2) Most shared files are read-only

With these assumptions, NFSv2's weak consistency isn't much of a problem.

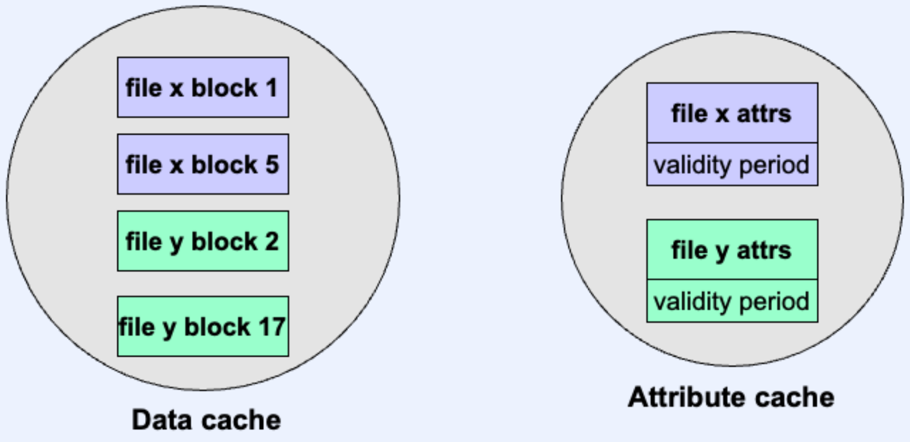


NFS Consistency

Achieving strict consistency in a distributed file system is difficult, particularly if performance is a concern. For a client application to use cached data with strict consistency, it must be assured that this data has not been modified by some other client. One approach for achieving this is for each client to check with server every time it accesses its cache to determine whether the cached block has been changed elsewhere.

The approach taken in NFS is to relax the requirement for strict consistency, allowing caches to be inconsistent, but with the benefit of performance gains. This makes sense because few applications are affected by such potential inconsistency: cases in which files are shared by multiple applications in a read-write fashion without some sort of additional synchronization are rare. Most applications either don’t share files or share them in a read-only fashion. What NFS does, resulting in **weakly consistent** caches, is to allow clients to use their caches for settable periods of time without checking with the server to see if the file has been modified.

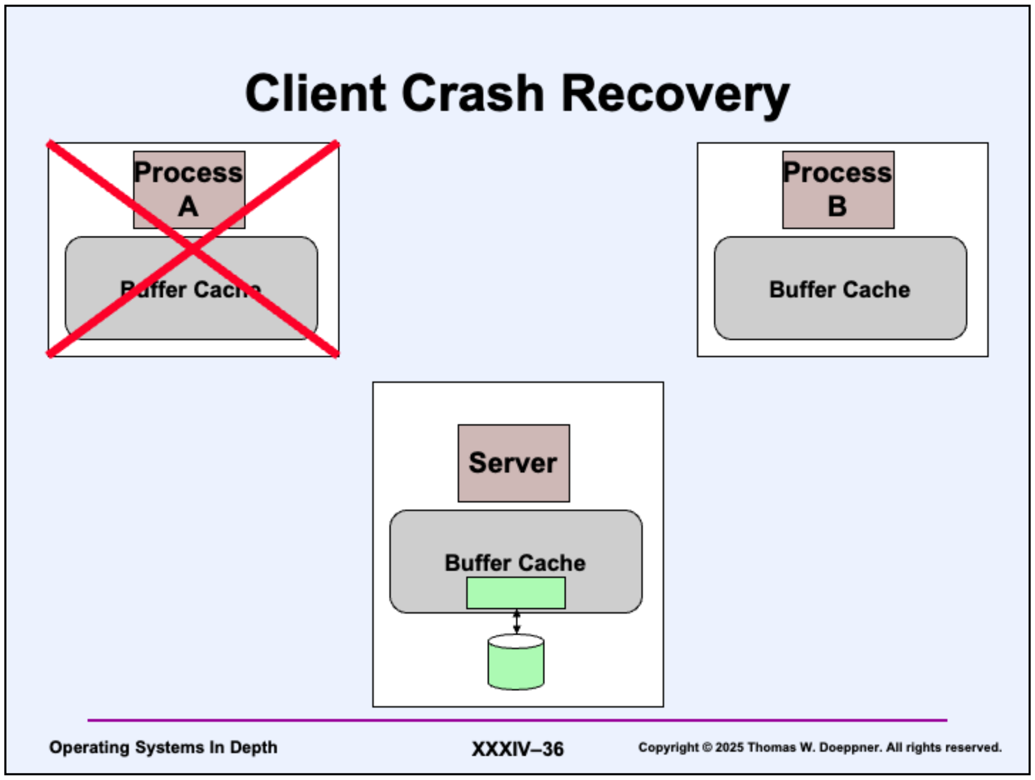
The Attribute Cache



The client may use the blocks in its data cache as long as the associated files' attributes are in the attribute cache and still valid. Once the validity period for a file's attributes has expired (their validity period is from a few seconds to a minute), the client must fetch new attributes from the server. If these show that that the file has been modified in any way, the file's blocks in the data cache are flushed and new blocks must be obtained from the server. Included among the attributes is file size and modification time.

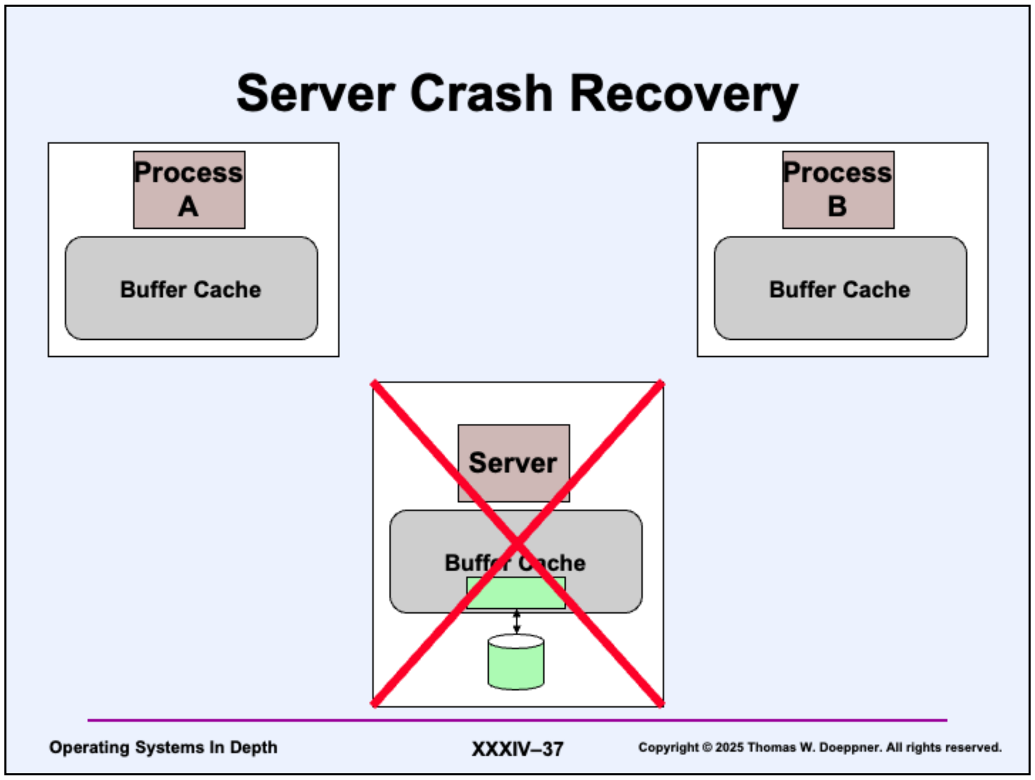
More …

- All write RPC requests must be handled synchronously on the server
- Close-to-Open consistency
  - client writes back all changes on close
  - flushes cached file info on open



Client Crash Recovery

Since NFS servers are stateless, the crash and restart of a client has no effect on them. Thus client crash recovery involves no actions on NFS's part.



Server Crash Recovery

Recovery from a server crash is easy for the server—it merely resumes processing requests. The client is generally delighted when the server recovers; its concern is what to do while the server is down.

Client machines may choose to "mount" NFS file systems in one of two ways: soft mounts and hard mounts. With a soft mount, if the server crashes, attempts to access the down file system fail—a "timed-out" error code is returned. This is often not terribly useful, since most applications are not equipped to deal with such problems. In the other approach (the hard mount) if the server crashes, clients repeatedly re-try operations until the server recovers. Thus, for example, a **write** system call to a down server will not return to the caller until the server comes back to life and responds to the request. This can try the patience of users, but it doesn't break applications.

Quiz 3

A client is modifying a file on the server, using a write RPC call (that specifies which file and where in the file). The file system is "hard-mounted". The server crashes, then, in a few minutes, comes back up.

a) It is not clear to the client application if its most recent write (before the crash) took place on the server

b) It is clear to the client application that its most recent write took place on the server

c) Its most recent write did not take place on the server

File Locking

- State is required on the server!
  - recovery must take place in the event of client and server crashes

Quiz 4

Can it be determined by a server that one of its clients has crashed and rebooted (assuming some cooperation from the client)?

a) no

b) yes with high probability

c) yes with certainty