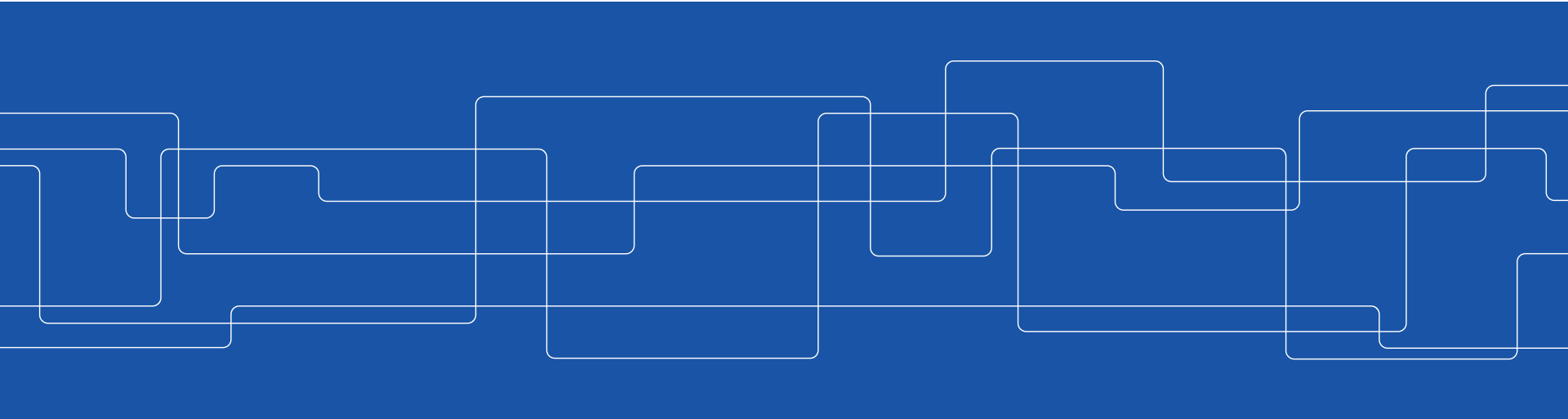
KTH ROYAL INSTITUTE
OF TECHNOLOGY

Distributed file systems

Vladimir Vlassov and Johan Montelius



What’s a file system

Functionality:

- persistent storage of files: create and delete
- manipulating a file: read and write operations
- authorization: who is allowed to do what
- a naming and directory service

The mapping of names to files is quite separate from the rest of the system.

Implementation

directory	map from name to identifier
file module	locate file, harder in distributed systems
access control	interacts with authentication system
file operations	read and write operations
block operations	
device operations	

What is a file

- a sequence of bytes
- attributes, associated meta-data
  - size and type
  - owner and permissions
  - author
  - created, last written
  - icons, fonts, presentation....

Unix file operations

- create(name, mode) *returns a file-descriptor*
- open(name, mode) *returns a file-descriptor*
- close(fd)
- unlink(name)
- link(name, name)

Can we separate the name service from the file operations?

Unix file operations

- read(fd, buffer, n) *returns the number of bytes actually read*
- write(fd, buffer, n) *returns the number of bytes actually written*
- lseek(fd, offset, set/cur/end) *sets the current position in the file*

- stat(name, buffer) *reads the file attributes*

Programming language API

Operating system operations are not always directly available from a high level language.

Buffering of write operations to reduce the number of system calls.

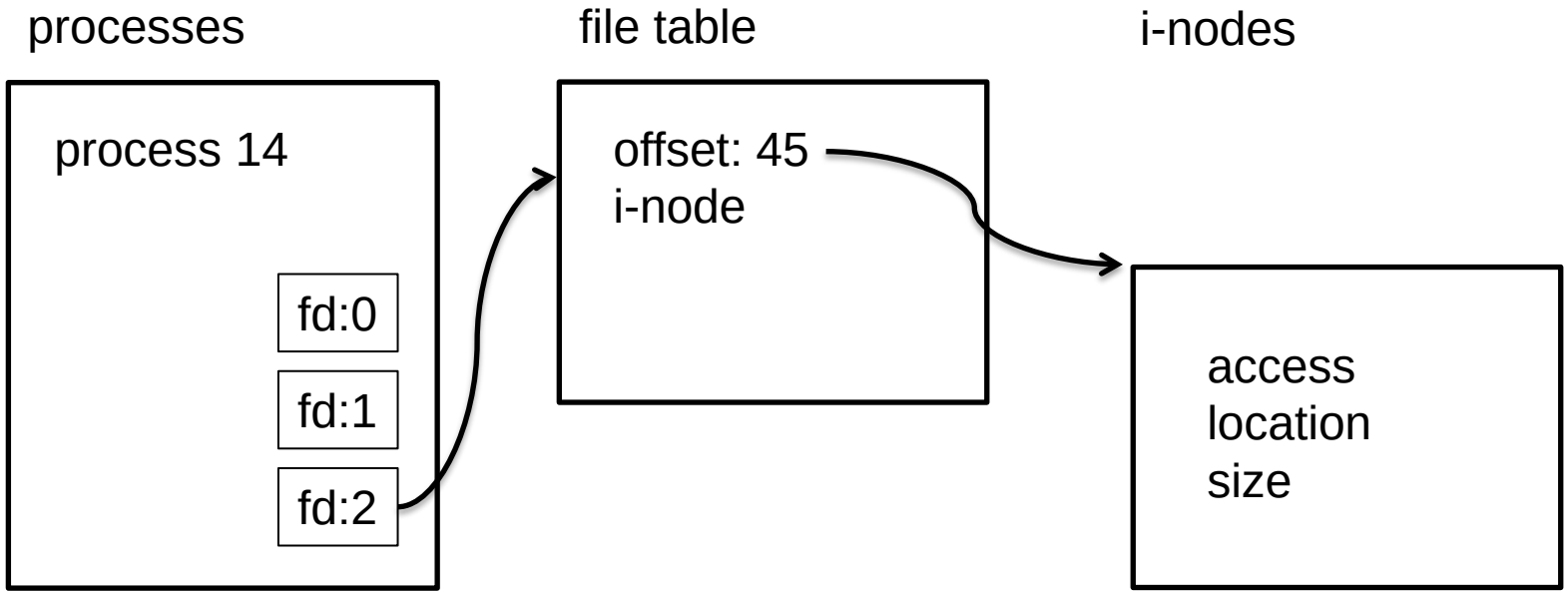
Descriptors, table entries and i-nodes

A process holds a set of open ***file descriptors***, each descriptor holds a pointer to a table of open files.

The file table entries holds a ***position*** and a pointer to an ***inode*** (index node).

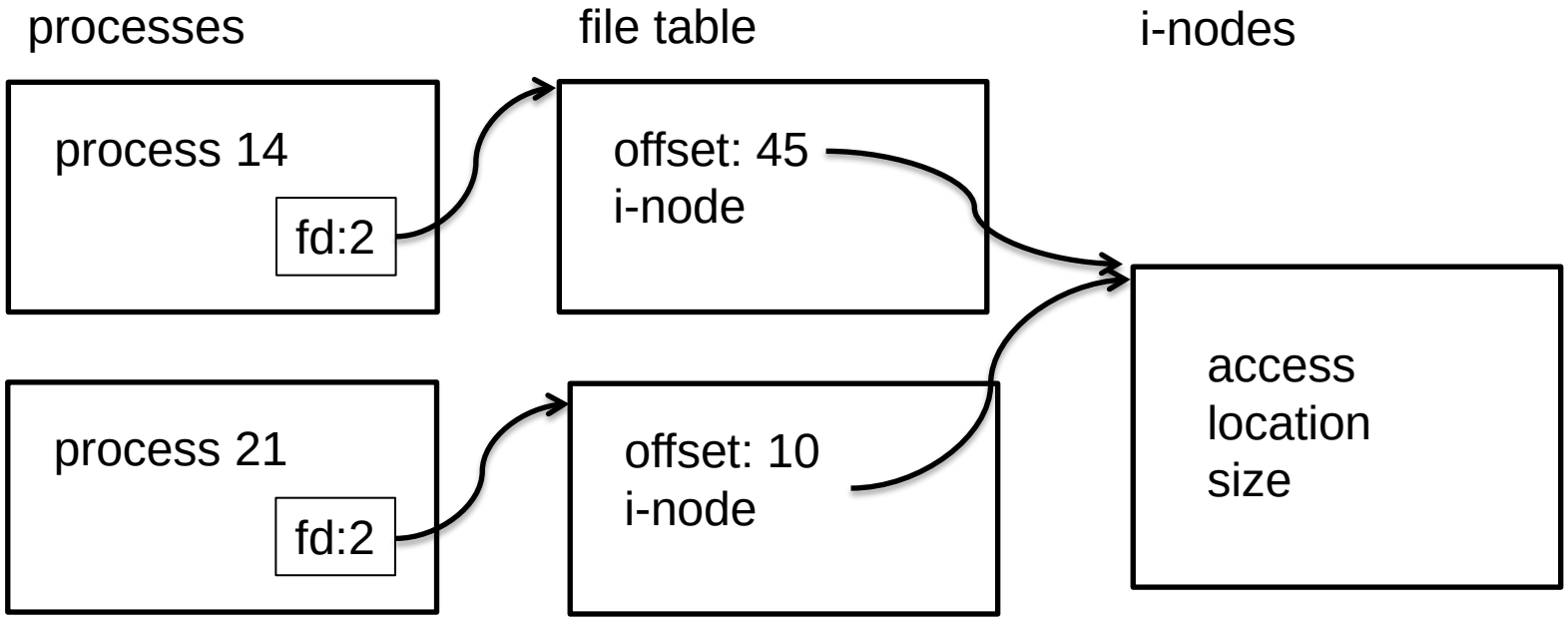
The inode holds information about where file blocks are allocated.

Descriptors, table entries and i-nodes



All *read*, *write* and *lseek* operations will change the current position in the table entry.

Two processes open the same file



Two processes that open the same file will have independent file table entries.

Nota bene

All threads in a process (or even if process is *forked*) share the same file descriptors and thus share the file table entry.

One-copy semantics

Most file systems give us a ***one-copy semantics***

- we expect operations to be visible by everyone and that everyone see the same file
- if I tell you that the file has been modified the modification should be visible

An architecture of a distributed file system

Let's define the **requirements** on a distributed file system.

Transparency - no difference between local and remote files

- access: same set operations
- location: same name space
- mobility: allowed to move files
- without changing client side
- performance: close to a non-distributed system

Concurrency - simultaneous operations by several clients

Heterogeneity - not locked in to a particular operating system

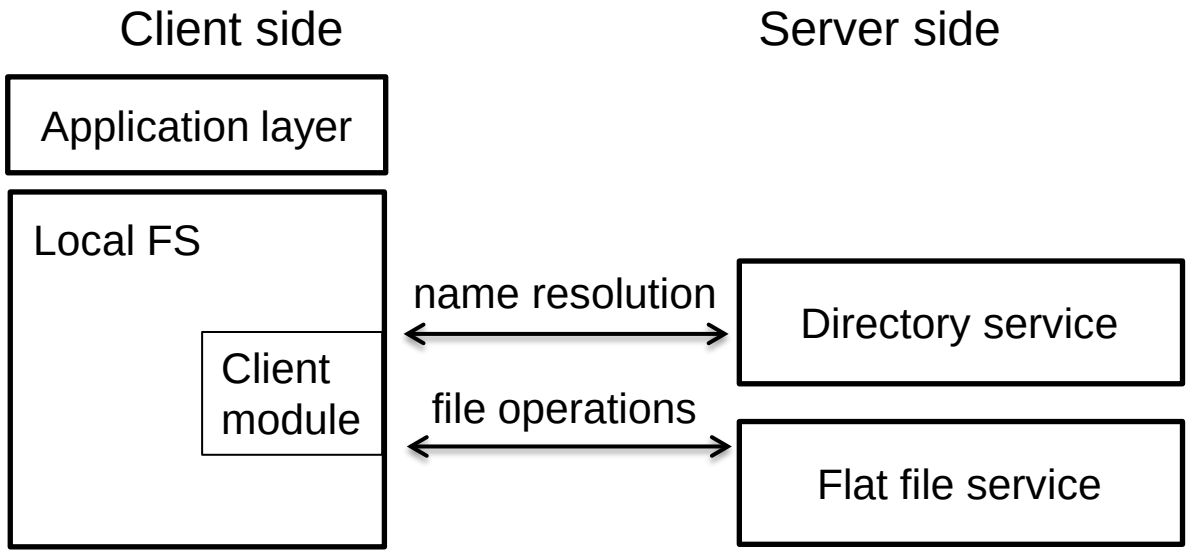
Fault tolerance - independent of clients, restartable etc

Consistency - one-copy semantics.... or?

Security - access control, who is allowed to do what

Distributed architecture

Separate the directory service from the file service.



The directory service

The directory service - what operations do we need?

- **lookup** a *file identifier* given a *name* and *directory*
- **link** a *name* to a *file identifier* in a *directory*
- **remove** a link
- **list** all *names* in a *directory*

The directory service does not create nor manipulate files.

The file service

What operations should be provided?

- **create** a file and **allocate** a *file identifier*
- **delete** a file
- **read** from a file identified by a *file identifier*
- **write** a number of bytes to a file identified by a *file identifier*

Do we need a **open** operation?

- What does open do in Unix?
- What do we need if we don’t have an open operation?
- What would the benefit be?

A stateless server

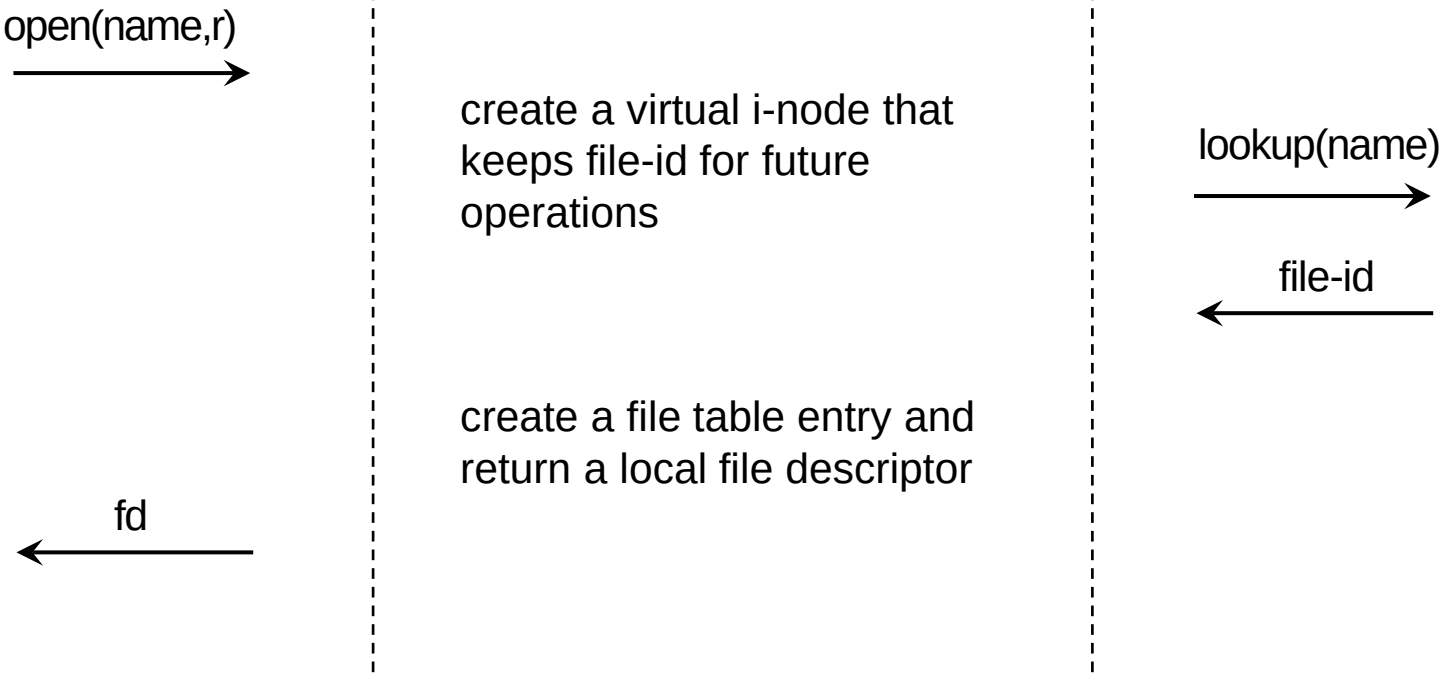
- What are the benefits of a stateless server?
- What are the benefits of a stateless server?
- How can we maintain a session state while keeping the server stateless?

How do we handle security?

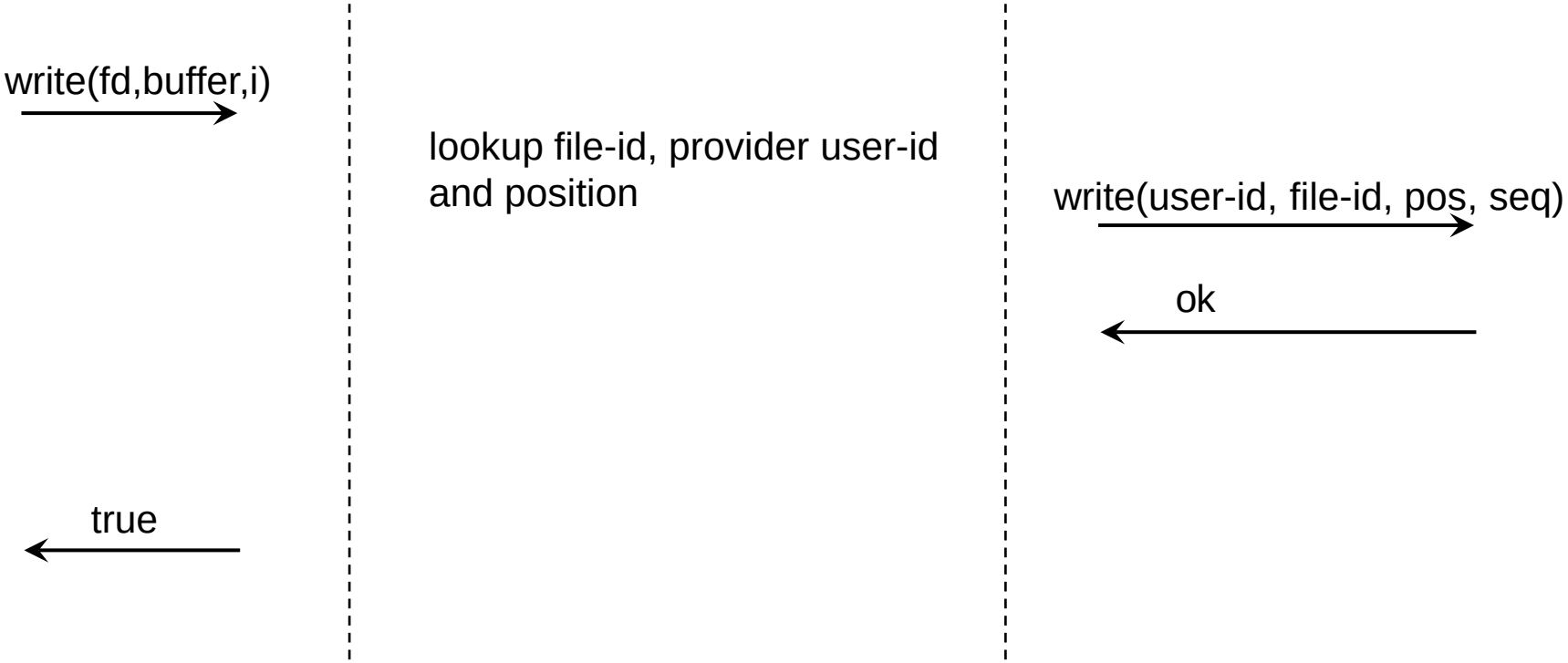
In Unix, permissions are checked **when a file is opened** and access to the file can then be done without security control.

If we do not have an *open operation*, how can we perform authentication and authorization control?

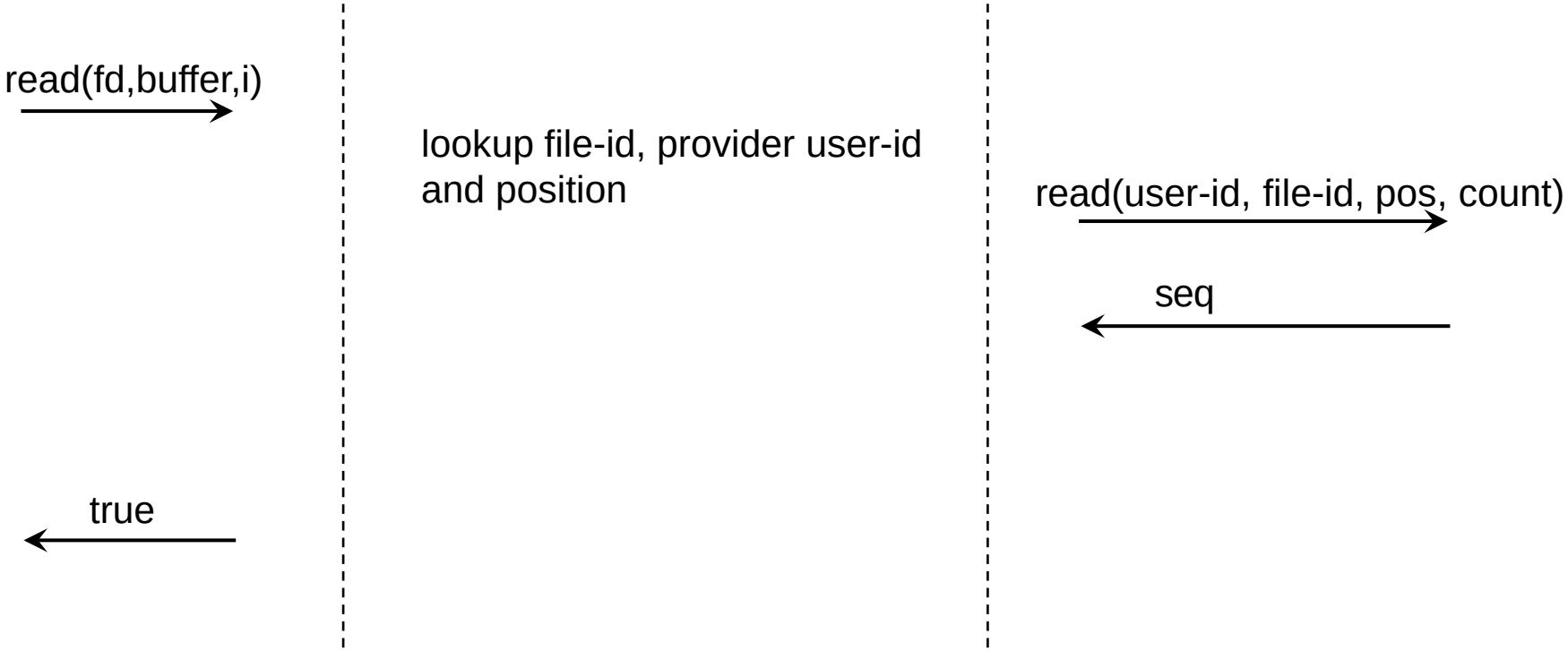
Client interface - open



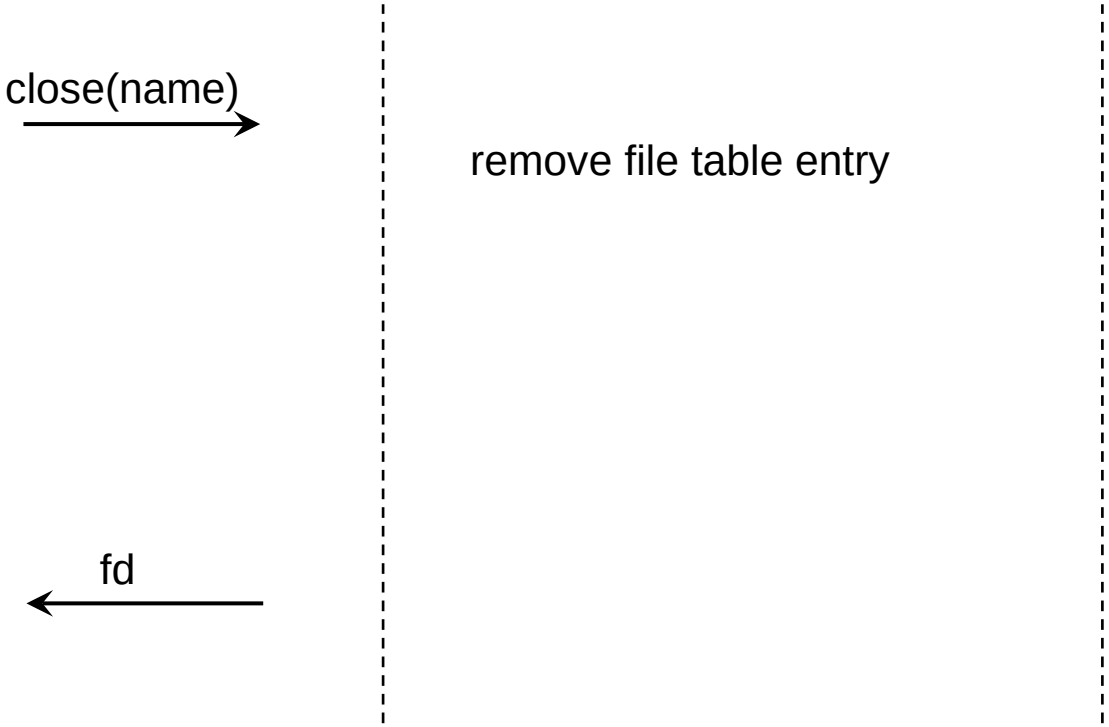
Client interface - write



Client interface - read



Client interface - close



Performance

Everything would be fine, if it was not for performance.

Keep a local copy of the file at the client side.

Cashing - options

Reading from a file: how do we know it is the most current?

- check validity when reading
- ... if you haven't done so in a while
- server should invalidate copy

Writing to a file:

- write-through: write to cache and server
- write-back : write to cache only
- write-around: write only to server

Caching could break the one-copy semantics

NFS - Network File System

- developed by Sun, 1984 targeting department networks
- implemented using RPC (Open Network Computing)
- public API: RFC 1094, 1813, 3530
- originally used UDP, later versions have support for TCP to improve performance over WAN
- mostly used with UNIX systems but client on all platforms available

NFS - client side caching

Reading from a file

- first read will copy a segment (8kB) to the client
- the copy valid of a time *t* (3-30 sec)
- if more time has elapsed, the validity is checked again

Writing to a file:

- write-back: write to the cache only
- schedule written segment to be copied to the server
- segment copied on timeout or when the file is closed (sync)

The server is stateless

AFS - Andrew File System

- developed by Carnegie Mellon University
- clients for most platforms, OpenAFS (from IBM),
  Arla (a KTH implementation)
- used mainly in WAN (Internet) where the overhead of NFS would be prohibitive
- caching of whole files and infrequent sharing of writable files

AFS - client side caching

Reading from a file

- copy the whole file from server (or 64kB)
- receive a *call-back promise*
- file is valid if promise exists and is not
- too old (minutes)

Writing to a file:

- write-back: write to the cache only
- file copied to server when closed (sync)
- server will invalidate all existing promises

SMB/CIFS

- ***Service Message Block (SMB)*** was originally developed by IBM but then modified by Microsoft, now also under the name ***Common Internet File System (CIFS)***.
- not only file sharing but also name servers, printer sharing etc.
- ***Samba*** is an open source reimplementation of SMB by Andrew Tridgell

SMB/CIFS client side caching

- SMB uses *client locks* to solve cache consistency
- A client can open a file and lock it; all read and write operations in client cache
- A read only lock will allow multiple clients to cache and read a file
- Locks can be revoked by the server forcing the client to flush any changes
- In a unreliable or high latency network, locking can be dangerous and counter productive

Summary

- separate directory service from file service
- maintain a view of only one file, one-copy semantics
- caching is key to performance but could make the one-copy view hard to maintain