A Cache to Bash for 9P

Geoff Collyer

Bell Laboratories
Murray Hill, New Jersey 07974
geoff@plan9.bell-labs.com

Charles Forsyth

Vita Nuova
www.vitanuova.com
forsyth@vitanuova.com

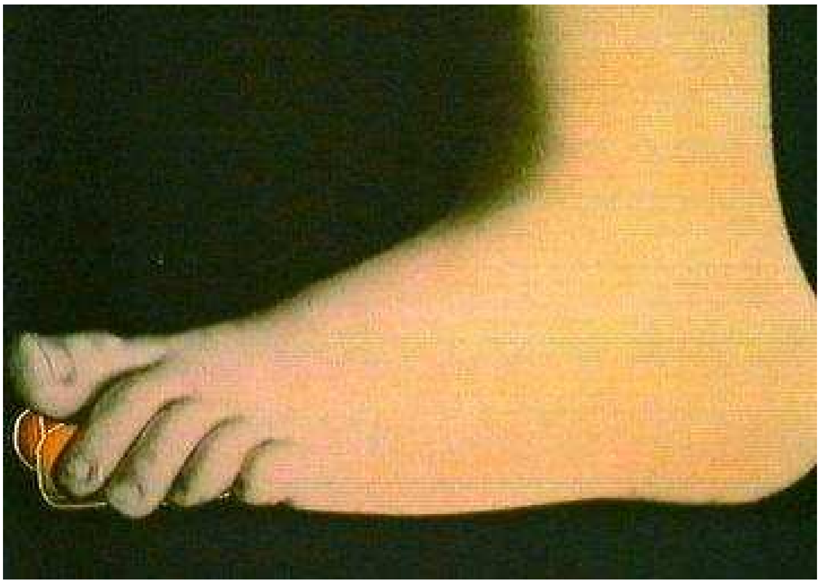
Why cache?

Plan 9 on Blue Gene:

— thousands of IO and CPU nodes (IO:CPU ratio is typically 1:32 or 1:64)
— scientific programs (eg, SPMD)
— system booting and initialisation

all attached to ...
... a single shared file server


OUCH!! BRAIN OVERLOAD!! IT'S TOO MUCH INFORMATION!!
PURGE! PURGE! PURGE!
© 2001 United Feature Syndicate, Inc.
www.dilbert.com



Can cache?

Many processes are doing roughly the same thing on a small, fixed set of files

Not massive files (eg, scientific I/O).
Not streamed data.
Not system services (synthetic files).

Essentially files providing infrastructure: eg, programs, libraries, configuration, parameters

It is important to cache *invalid* accesses.
(Example: Python’s search for libraries ⇒ fine for one instance, but 17 hours for thousands!)

Cache structure

We modelled it on existing *cfs*:

FS ↔ *C*ache ↔ *FS*

A user-level program acts as a file server to its clients, and a client to a remote file server, providing the cache in between.

It transforms a single stream of 9P traffic.

It answers the clients itself, whenever it can.

When it cannot answer, it

delegates the request to the server
returns the reply to the client
also caches the result.

Meta-data is cached as well as data (unlike *cfs*), including path names.

9P requests

The work to do is defined by the set of 9P requests:

`Tversion` *tag msize version*	start a new session
`Tauth` *tag afid uname aname*	optionally authenticate subsequent attaches
`Tattach` *tag fid afid uname aname*	**attach to the root of a file tree**
`Twalk` *tag fid newfid nwname nwname*wname*	**walk up or down in the file tree**
`Topen` *tag fid mode*	**open a file (directory) checking permissions**
`Tcreate` *tag fid name perm mode*	**create a new file**
`Tread` *tag fid offset count*	read data from an open file
`Twrite` *tag fid offset count data*	write data to an open file
`Tclunk` *tag fid*	**discard a file tree reference (ie, close)**
`Tremove` *tag fid*	remove a file
`Tstat` *tag fid*	retrieve a file's attributes
`Twstat` *tag fid stat*	set a file's attributes
`Tflush` *tag oldtag*	flush pending requests (eg, on interrupt)

How must the cache respond to each?

What data types are needed?

Fid handling

A fid represents an active file, and we aim to reduce fid usage on server (hence file descriptor usage), with many client fids sharing a single fid on the server.

There are two sets of fids:

- one built by the client processes (each actually an *exportfs* representing many clients on a CPU server), managed by the client
- one set representing active files on the server, managed by the cache

Fscfs must map from the first set of *Fids* to the second set of *SFids*.

Data types

IOmode ::	R | W | RW
Fid ::	fid: u32int qid: Qid path: Path opened: SFid mode: IOmode
SFid ::	fid: u32int
Path ::	name: string qid: Qid parent: Path kids: set of Path (Valid | Invalid)
Valid ::	sfid: SFid file: optional File
Invalid ::	reason: string
File ::	open: IOmode↦SFid dir: Dir clength: u64int cached: sparse array of Data
Client ::	fids: u32int↦Fid root: Path

Client requests contain integer *fids*, that are mapped to *Fids*, that refer to the *Path* tree node resulting from a walk from the *root*.

Active Fids for the same Path *share* an integer fid referring to that path on the server.
That is found by the *SFid* stored in the Path; the *SFid* itself is shared.

Building the Path

1. `Tattach` *fid*
   Delegate to the server, replacing the incoming (local) fid by a new server fid.
   Create an empty Path tree for the *root*, associated with the new SFid. The Path is also associated with *fid*.
2. `Twalk` *fid* $s_0 \cdots s_{n-1}$
   Start from Path associated with *fid*, and attempt to walk the sequence of names.
   If the walk succeeds locally, and the resulting Path has an SFid, reply to the client.
   If the walk fails with an Invalid entry, return an appropriate error.

   Otherwise, delegate to the server — walking to a *new* fid on the server. Update the Path tree (add each successful s_i to the tree, and record an Invalid entry on an error).
3. `Tcreate` *fid name* $\cdots$
   There's a directory Path associated with *fid*.
   Delegate to the server, using a *clone* of that Path's SFid.
   Add a new child Path associating *name* and *fid*, where *fid* is now open on that new SFid.
4. `Tclunk` *fid* clunk the corresponding *fid* locally (discards the Fid), and reply to the client.

Paths and SFids are reference counted: clunking Fids locally might result in release of SFid and clunk of its server fid.

Results

When many processes are making identical file system requests over a given interval, *fscfs* aggregates them into single requests at the server.

On an IO node, for an interval from initial connection to the file server, until its 64 CPU clients were ready to go:

Op	*IO node*	*Server*
`Tversion`	1	1
`Tattach`	1	1
`Twalk`	7,855	56
`Topen`	1,486	77
`Tread`	6,823	133
`Tclunk`	4,749	0
`Tstat`	4,224	4,224
bytes read	19,913,992	462,722

Ron can run his benchmarks, and a new sad tale begins!