Parallel File System

Farshid Farhat & Diman Zad Tootaghaj

Assumptions:

Clients (CLs) and file servers (FSs) know the (IP,PORT) of meta-data manager (MM). Therefore file servers export their (IP,PORT) to MM, and MM imports them to any connected CL. We put predefined MM's (IP,PORT) in config.h files of all projects (PFS_client, PFS_MM, and PFS_server). Our codes don't have any warnings and all warnings are related to test codes!

We added one function at the beginning of the test codes called "initialize(argc, argv);"! It's necessary to initialize the client to make sockets with MM & FSs. For modularity it was better to separate initialization step from pfs_create (not inside this function), because it may call pfs_create multiple times.

All 'w' in test codes should be changed to "w"! Also I put "argc=2;argv[1]="1";" at the beginning just for debugging the code that could be omitted and compile again.

Also it is assumed that the test-c1.c is first executed and then test-c2.c, test-c3.c, test2.c and test3.c can be run, because test-c1.c creates the pfs_file1 that can be used by other test codes. Also "file" string inside all test codes was changed to "pfs_file1" to be compatible with the first test code (test-c1.c).

Instructions to RUN PFS

First you have to set MM's (IP,PORT) in all config.h. If you want to run it in the same machine, it's OK as it set to (127.0.0.1,10000). Then you should compile the codes with gcc … –lpthread or by using Makefile. Then you can run them as MM, FSs and then Clients.

For local machine test directly you can run Debug/PFS_MM to initialize and make MM listen to its file server export (IP,PORT). Then you can run Debug/PFS_server one by one (multiple times) to export their (IP,PORT) to MM. After all you can run any client by Debug/PFS_client after compile with your test code.

To run client with a different test code, you should only compile it with that code inside the directory. Then you should follow above instruction. Also there is a Makefile inside each project (client, MM and FS) that should be used for compiling.

The functions harvester(), flusher(), save2cache(), readcache(), MMR() and release() are dealing with Buffer Management (Client Cache Control) in the client. Respectively harvest free/used space, flush dirty blocks, save a block in direct-mapped cache, read a block from direct-mapped cache with LRU update, get any lock-releasing message from MM asynchronously and release the lock and send to MM.

R/W Tokens are managed in MM inside readfile(), writefile() and release() functions. If a client releases its lock after any write, MM will broadcast invalidation message to other clients asynchronously.

Design Choices

1. MM is fully multi-threaded and manages consistency by using mutex_lock and tokens, so FS is only multi-threaded and don't need mutex_lock as MM takes care of that. Clients are most of the time one thread but when MM wants to invalidate their cache, they have to receive invalidation message asynchronously.
2. Token management exploits a dynamic array of each file blocks to keep clients' accesses to that file. This mechanism solves prior issues mentioned by Prof. Anand and TAs. Amazingly it's not necessary to fuse or fission and can manage more complex scenarios but it exploits more memory.
3. Not only clients do their job (create/open/read/write/delete) sequentially but also they can receive asynchronous messages from MM to invalidate their cache and give back their tokens. It makes them semi-multithreaded!
4. The functionality of different parts of the codes has been commented inside the project implementation. All message communications are shown in text to be easily tracked and seen by an observer. We did text processing to handle this feature for convenient.